Ну и теперь переходим ко второй части Марлезонского балета. И попытаемся понять, для чего нам нужен Docker SDK в golang-е. И как мы будем с ним работать. И что, вообще, там нужно.

Во-первых, мы можем организовать тестирование с помощью Docker SDK. То есть локальное тестирование, естественно. В чем фишка. У вас есть приложение, работающее с базой данных, работающее еще с какой-нибудь Kafka-ой и еще какое-нибудь, дергающее внешний сервис. И вам нужно его протестировать. В нормальной реальности вы пишете либо интеграционный тест, то есть который там поднимает Embedded базы данных прямо внутри языка. И Embedded, все. Мокаете там ваш сервис и пишете. Это не очень удобно, потому что не получается end-to-end. Там вот эти Embedded базы, они может – что-то сохранят, может – не сохранят, может – это просто заглушка с интерфейсом.

Можно сделать Docker compose и запускать. И опять же, Docker compose тулза написана на golang-е, использующая Docker SDK, то есть уже как бы попадает в эту категорию организации тестирования. Но смысл в том, что она, например, не поддерживает запуск сценариев. То есть вы запускаете Docker SDK, потом у вас есть какие-то конкретные отдельные взятые написанные сценарии, и это надо все вместе запускать. Поэтому можете сделать свою тулзу, которая будет поднимать ваш контейнер с приложением. Опять же, может быть, с помощью Docker compose. Это не важно. И запускать тесты автоматические.

Соответственно, можно собирать с помощью Docker SDK данные о запущенных контейнерах. Вы можете там в ультимативной форме сделать у себя некое подобие Docker swarm или kubernetes. Потому что они-то тяжелые достаточно ребята, и они поддерживают много фичей, которые вам не нужны. Можете написать там свою фичу, сделать какой-нибудь routing для ваших тестов. И спокойно с ними играться. Организовывать можно процессы локальные, можно

организовывать процессы разработки. Опять же, Docker compose – это яркий пример организации именно процессов разработки. И соответственно, последнее, что вы можете сделать – это если у вас есть какой-то CI/CD pipeline, вы можете с помощью вашей тузы…, можете его еще и упростить. Давайте разберемся с каждым из этих пунктов по порядку.

Ну и давайте поговорим про организацию нагрузочного или интеграционного тестирования. Ну «или», в смысле это не одно и то же. Там я уже думаю, что некоторые мои зрители пошли за таблетками, потому что дед опять про свое нагрузочное тестирование говорит на локальной машине для разработчиков. Ну, короче, о чем я. Вы можете использовать, вполне спокойно тестировать нагрузку, потому что, как правило, ваш контейнер, он там будет запущен, ну там не знаю, 256 мегабайт памяти и половина CPU. Просто, потому что лучше иметь больше контейнеров.

И вы спокойно можете протестировать один (НРЗ 02:49) с вашего приложения в этих условиях. И сказать, что вот его пропускная способность, скажем там, 100 запросов в секунду, а потом он упирается в базу. Или там: его пропускная способность, там не знаю, 250 запросов, потом мы начинаем падать в каком-то методе. Это вполне спокойно, это доступно можно протестировать локально, пробенчмаркать. И помимо этого, вы можете автоподнимать и сканировать контейнеры, вы можете сделать несколько (НРЗ 03:18) вашего приложения, чтобы посмотреть, где у них узкие места.

Например, у нас очень часто сетевой обмен – это узкое место, потому что сами приложения делают не очень много работы по enrichment-е, по вгрузке данных, но достаточно много работы по сетевому обмену. И поэтому нам важно тестировать именно метрики сетевого взаимодействия, смотреть, где у них горячие точки. Это удобно можно сделать локально достаточно. Вместо того, чтобы что-то там

куда-то вгружать, запускать это все, снаружи смотреть. Гораздо больше возможностей есть локально. Например, можно профилировщики подключать и так далее.

Мерить метрики контейнеров. Опять же, CPU, память, входной трафик для каждого контейнера, строить какие-то там для себя графики, можно поднять, не знаю, локальные (HP3 04:09), что-то туда загрузить, можно не поднимать, можно просто строить графики для дальнейшего анализа, какие-то логи. Можете запускать какие-то сценарии, опять же, адаптировать benchmark, которые у вас есть в директории, либо сделать какую-нибудь там тулзу для benchmarking-а. Как правило, идут более простым путем, там просто пишут эти сценарии. И, опять же, мерить нагрузки на сеть, в принципе, то же самое, что и замер метрик, но вы можете, опять же, сделать какой-то сетевой адаптер и через него гонять данные. Смотреть, что, где болит у сетевого адаптера.

То есть довольно-таки мощный инструмент. Вместо того, чтобы базироваться на метриках отдаваемых Kubernetes-ом несмотря на то, что они хороши достаточно. Или каким-то облачным провайдером, который еще с вас потом чарджнится. То, что вы там нагрузку создаете, вы можете это сделать локально в каком-то маленьком масштабе. Всей картины оно не даст, конечно. Но какое-то интеграционное тестирование, локальное тестирование можно сделать с помощью написания простых довольно-таки скриптов на golang-е и раздачи их локально разработчикам.

Теперь про какие локальные процессы я говорю. То есть что это значит – организация локальных процессов. Во-первых, вы когда запускаете локальный контейнер, у вас должно организоваться какой-то проксирование redirect трафика. Как правило, в большинстве случаев, которых я видел, просто организуется общая сеть через тот же Docker compose. И в ней все общаются со всеми. Возможно, что

в вашей ситуации это может быть не так, организуются какие-то локальные сегменты. Это вы можете проэмулировать с помощью Docker SDK как раз. Либо вы хотите сделать (НРЗ 05:49), когда у вас отваливаются какие-то сегменты сети. И при этом, вам нужно продолжать работать.

Существуют, опять же, всякие тулзы от Netflix-а, по-моему, называется Chaos Monkey, как раз. И существуют (НРЗ 06:04) механизмы. Но локально если вы хотите быстро проверить, там довольно простой скрипт на golang-е, в котором вы указываете…, в котором можете указывать сегменты сети, их просто включать или выключать с какой-то периодичностью. В момент запуска тестов, например. Очень удобно. Можно сделать какие-то локальные чеки, например, что ваше приложение пишет внутрь контейнера, какие..., как при этом меняются метрики, которые приложение дает. То есть тоже довольно удобно. То есть вам не нужно подгадывать моменты, тем более связывать две тулзы. Вы запустили какой-то тест, с ним вместе стянули метрики, в тот же момент или параллельно в отдельной горутине. И вполне спокойно проверили.

Можно анализировать логи, которые приложение ваше отдает на предмет каких-то логов, которых быть не должно. Например, мы так ловили довольно много ошибок. Просто, смотря на какие-то трейсы. То есть у нас приложение автоматически определяло, есть ли у вас логи уровня error, в каком они, значит, месте, и были ли они до этого. То есть у нас была какая-то база.

Соответственно, опять же, есть автоматические инструменты для этого анализа, можно в том же (НРЗ 07:23) это сделать. Но, опять же, локально довольно сложно. Можно перезапускать контейнеры, чтобы посмотреть, как ваше приложение при этом себя ведет. Опять же на каких-то тестах. Например, если у вас какое-то сетевое приложение, то, что произойдет, если нет ни одного контейнера

запущенного. Вернее, мы, в принципе, знаем, что произойдет, но, опять же, насколько быстро оно включится в работу. Это важная метрика.

Можно перезагружать конфигурации, смотреть, как при этом реагирует ваше приложение на их перезагрузку. Опять же, тоже важная часть. Понятно, что в production-е у вас будет Kubernetes, у которого есть эти самые ConfigMap-ы, которые перезагружают environment и перезапускают контейнеры. Вы можете спокойно сэмулировать локально, и прекрасно у вас все будет работать. И, опять же, я уже упоминал про это. Запуск всяких баз данных, Kafka-ок, Rabbit-ов, всяких там инфраструктурных тулз, возможно, каких-то LDAP контейнеров, что угодно, что не относится к вашему приложению – вы можете положить, соответственно, рядом и запускать это как инфраструктуру для ваших тестов.

Ну и что же такое упрощение pipeline-а. Опять же, вы можете в ваш скрипт интегрировать в систему CI/CD. Для того, чтобы какие-то вещи там делать. Почему бы и нет. Во-первых, вы объединяете шаги. То есть в системах CI/CD очень часто можно встретить…, ну поскольку, мы оперируем там обычно на уровне каких-то системных тулзов, разделение такое достаточно (HP3 08:54), что в этом моменте мы вызываем там docker run. Потом мы собираем, значит, с него какую-то информацию, потом мы, значит, получаем список, из списка вызываем еще что-то. Ну и так далее. В общем, это можно и нужно объединять, довольно просто, если у вас, например, есть контейнер, который вы запускаете, собираете с него логи 10 секунд. Потом, значит, гасите контейнер. И передаете логи дальше. Это будет четыре шага. В golang-е вы можете это сделать в один шаг. Написать маленькую тулзу. И положить ее просто в CI. Просто вызвали тулзу, она все сделала, вы собрали логи, пошли дальше.

Логические условия и циклы. Опять же, у разных систем CI с этим по-разному. Опять же, у GitLab-а хуже. У, например, Jenkins-а лучше из (HP3 09:42). Но в

целом довольно больно. Опять же, системы CI не особо предназначены для кодинга в них. Это не языки программирования, поэтому ну, естественно, они будут проигрывать каким-то вашим скриптам.

Контроль доступа. Опять же, если вы связываете систему CI с каким-то контрольным доступом, например, LDAP, где вам нужны какие-то конкретные разрешения на подключение, скажем, к базе данных в облаке, если вы там тестируете ваш интеграционный тест, то есть какой-то staging базы данных в облаке, при этом запуская все остальное локально, то вам нужен, соответственно, какой-то контроль доступа. Опять же, иногда бывает, что в системах CI нужно проксировать пользователя, пересоздавать пользователя, ну в общем довольно геморройный процесс. В golang-е вы можете просто вызвать LDAP контейнер. И соответственно, использовать просто напрямую вашего пользователя.

Взаимодействие с API. Опять же, как правило, системы CI работают на уровне каких-то конкретных утилит, системных утилит командной строки. И поэтому взаимодействие с API у вас будет настолько, так сказать, богатым, насколько вам позволяют эти системные утилиты. В языках, как правило, можно найти какие-то workaround-ы. Вы не ограничены тем, что автор заложил в эту системную утилиту, вам не нужно никаких обновлений, вы просто берете и пользуетесь всеми возможностями API, не говоря уж о том, что, как правило, это удобнее.

На этом моменте меня, скорее всего, второй раз отправят за таблетками, потому что скажут: «Дед, мы там на «Питоне» автоматизируем, в чем проблема, Bash-скрипты у нас есть, всякие тулзы». Да, конечно, можно работать и с помощью Bash-скриптов, сделать то же самое. Опять же, можете в Groovy сделать все вот эти вот этапы. Опять же, не понятно, как это делать локального пользователя.

Ну короче, смысл в том, что вы можете то же самое повторить либо на Bash-скрипте, лучше сделать скрипт на Python, потому что Python – это язык программирования, у него есть обвязки, у него есть Docker SDK и так далее. Но тут есть по сравнению с golang-ом, опять же, golang не идеал, у него есть свои проблемы, но здесь есть парочка моментов, про которые надо подумать перед тем, как использовать именно вот эти вот пути автоматизации.

Какие проблемы у нас с Bash-скриптом. Ну, во-первых, мы ограничены тулкитом. Опять же, Bash-скрипт – не язык программирования. Вы можете с помощью cURL вызывать API напрямую, например, если у вас нет какого-то..., каких-то там инструментов. Ну вы можете там себе представить размер этих скриптов, с учетом вообще самого Bash-а, самой структуры Bash-а, это будет довольно-таки сложно. Нужно, опять же, ставить Docker SDK, нужно ставить все эти приложения, нужно ставить SSL утилиты, ну не знаю, мне кажется, что это довольно большой объем. В принципе, для локальных пользователей, скорее всего у них и так все это будет на компьютере стоять, в принципе, ничего плохого в этом нет. Какие-то вещи в Bash-скрипте сделать проще, чем писать код в golang-е, потому что он будет менее избыточный. Но, тем не менее, что-то то придется ставить.

Разные системы Bash SSH, например, в MacOS по умолчанию. Где-то просто SH есть, какие-то конструкции не поддерживаются, какие-то поддерживаются. Если там держать в голове, на разных системах это может не завестись, придется писать под каждую систему отдельно. Ну, короче, есть определенная накладная работа по maintenance-у этого всего, по поддержке этого всего под разные системы пользователей. И можно ее избежать. И опять же, сложно писать большие pipeline-ы из-за структуры самого языка. Потому что Bash-скрипт предназначен для просто склейки нескольких вызовов с функции системных утилит и обработки их выводов. И все. И то есть циклы, (НРЗ 13:53), это все по остаточному принципу делать. Это не очень удобно. На мой взгляд.

Ну и какие проблемы у нас могут быть с «Питоном». Опять же, «Питоном» все намного лучше, есть и обвязки, есть и Docker SDK, есть и все остальное. Вот вы написали такой счастливый набор скриптов на Python, пришли к разработчикам и говорите: «А все, ребята, теперь вы ставите Python, ставите Pip, в общем, качаете все эти зависимости, запускаете скрипты». Они такие: «Блин, что мы вообще там на плюсах пишем, на Java-е пишем, какой, блин, «Питон», зачем нам это все». Можно, конечно, собрать приложение в большой бинарник. Подчеркиваю слово большой. Будет большой. Потому что с собой все зависимости тянет вместе с обвязками.

Ну, в общем, tooling, делать tooling на Python. Ну можно, даже будет работать, будет хорошо. Но, если вы…, такая одна команда, использующая Python, наверное, лучше подумать о чем-то более системном, что потянет свои зависимости и просто будет запускаться у разработчика в системе.

«Питон» в отличии от golang-а не поддерживает горутины, у «Питона» концепция (HP3 15:04). Такая же, как…, ну, вернее, она не такая же, она несколько менее развита, чем в Java-е. Но в общем, синхронизацию разных потоков довольно сложно поддерживать. А в тестах, как правило, ее надо синхронизировать. То есть в приложениях для тестирования самым простым, конечно, является запуск потока, ждать 10 секунд, поток вывалится, сам тебе об этом скажет. Но если вам нужно, например, сделать запрос, в этот же момент его померить, чтобы первый поток подождал, пока второй поток его померит и так далее. Ну в общем, там начинаются приключения, которых можно избежать с golang-ом. И которых нельзя избежать с Python-ом. Опять же, ни один из этого не (HP3 15:45). Используйте, что вам нравится, что вам удобно. Просто в golang-е эти моменты покрыты лучше.