Так, давайте посмотрим на базы данных, которые у нас сегодня будут. Соответственно, я создал файлик docker-compose. Файлик, который объединяет в себе несколько Docker-контейнеров с различными базами данных. Удобно для запуска и отладки. Соответственно, начнем мы с базы данных postgres.sql, базы данных общего назначения, реляционной базы данных из первой категории.

Потом посмотрим на базы данных ключ-значение redis. И посмотрим на документно-ориентированную базу mongo.db. Не обязательно в этом порядке. Скорее всего, мы посмотрим сначала на postgres, потом на mongo. И сравним, как это все работается с redis-ом.

Ну что же, довольно прелюдий, поехали к postgres-у. Ну и давайте разберемся теперь, непосредственно, с postgres. У меня есть клиент postgres-а, встроенный прямо в мою IDE, в IDE Golang, который позволяет, собственно говоря, соединяться с базами данных. И позволяет, ну, соответственно, брать все структуры.

Что мы можем здесь увидеть? Что у нас есть базы данных. База данных postgres. Это такая стандартная база данных по умолчанию, такое у нее имя. В ней есть всякие там расширения, языки внутренних функций, методы. Нас это не очень сейчас интересует. Нас интересуют схемы. Схемы бывают публичные, то есть у них есть NewSpace-ы свои. В NewSpace-ах у нас лежат таблицы. И вот таблицы – это и есть хранилище нашей информации.

То есть представим, что у нас есть задача. У нас есть пользователи в компании, в какой-то. Какая-то есть компания, у нее есть парковочное место. И нам нужно, соответственно, в этом парковочном месте хранить машины. И пользователи, которым принадлежат эти машины. То есть достаточно, на самом деле, двух таблиц. У нас будет таблица пользователей. То есть в ней есть Id – это

обязательное поле вообще для баз данных. Ну то есть вы можете создать таблицу без поля Id, но у нее будет какое-то внутреннее Id, потому что это важный элемент для определения, соответственно, где и как лежит столбец.

Соответственно, есть такой вот набор полей. Вот этот самый идентификатор, есть имя, есть должность этого пользователя или ранг его в иерархической структуре, и есть ключ. То есть, что такое ключ? Ключ – это что-то, что уникально идентифицирует конкретную запись в таблице. То есть, если у вас есть 100 записей в таблице, у вас должно быть 100 записей, принадлежащих первичному ключу. И соответственно, это позволяет другим таблицам ссылаться на этот первичный ключ. То есть если вы хотите установить какую-то связь с пользователями, вы можете использовать первичный ключ для этого. Есть еще индексы и ограничения.

То есть здесь мы видим, здесь все, что у нас принадлежит первичному ключу. То есть ограничение на уникальность и, соответственно, индекс на первичный ключ. И есть вторая таблица – таблица с машинами. Здесь у нас идентификатор есть, соответственно, user_id. И вот здесь мы видим такой синенький ключ, и мы заодно видим, что есть fk. Так называемый foreign key – удаленный ключ.

То есть это значит, что user_id у нас ссылается на таблицу user, как раз ссылается на первичный ключ таблицы user, то есть на ключ id. И соответственно, здесь мы видим, да, что еще есть какие-то цвета, бренды и так далее. Если открыть таблицу пользователей, то мы видим таблицу пользователей, то мы видим здесь имена, соответственно, ранги и id-шники от 1 до 10, просто последовательно.

Если мы откроем таблицу машин, мы видим, что здесь уже все немножко сложнее. У нее тоже есть id¸ тоже первичный ключ. Есть id пользователя, то есть id пользователя, которому принадлежит эта машина, цвет, бренд и, соответственно,

номера. То есть пользователю 1 принадлежит четыре машины, как мы видим. Пользователю 2 принадлежит три машины, пользователю 4 – четыре, и так далее.

В принципе, и сейчас это запоминать не надо, мы с этим разберемся последовательно. Но как бы здесь просто надо понимать про представления. Да, это просто плоская таблица, в ней есть записи, в каждой записи есть какие-то атрибуты одинаковые в рамках данной таблицы. Соответственно, эти таблицы могут быть связаны. Потому что, например, если у вас один пользователь имеет несколько машин, то вы никак это в табличном форме не отобразите. Только как-нибудь через запятую можно писать или что-то в таком духе, что не очень удобно.

Вместе со всем этим в голове давайте уже посмотрим, как делаются, так называемые, запросы в эту базу данных для добавления и извлечения записей. Давайте посмотрим уже на пример Golang-а. Соответственно, что мы здесь видим? Мы здесь видим команду connect. То есть в языках программирования вообще, и в Golang в частности, вам не нужно думать о том, как там к базе данных подключаться, потому что база данных – это, на самом деле, она просто висит в инете, в сети какой-то. Она принимает запросы в определенном формате. И отдает запросы в определенном формате. То есть это обычный такой веб-сервер с каким-то API. И каким-то своим протоколом. Но из плюсов вам не нужно, как правило, думать о подключениях, об их открытии, закрытии, об удержании подключений, о каких-то специфичных фишках работы с базой.

Под все языки уже созданы обычно, так называемые, драйверы. То есть штуки, которые позволяют нам подключаться к базе данных. Так называемые, драйверы баз данных. И в данном случае в Golang-е их несколько. И я взял один из самых популярных, pgx. Если вы в Google-е забьёте, как подключиться к postgres Golang, это будет, наверное, третья ссылка. Не знаю точно. Соответственно, она

достаточно проработанная. У нее есть много фич, она используется в связке других, соответственно, (НРЗ 06:22), о которых поговорим немножко позднее. И она полезна для, значит, установки соединения.

Что здесь происходит? Мы, значит, передаем ей contex, в данном случае Background, потому что это не важно, какой у нас contex. У нас здесь просто приложение, здесь простая функция, которая заберет нам данные из базы, их покажет. И передаем строку подключения. Строка подключения определяется как url, то есть как обычный адрес веб-сайта, но с форматом «postgres:», вместо http или https. Затем у нас идет логин, пароль пользователя. И через @ у нас идет, соответственно, сервер.

То есть если он был бы где-то в инете, это был бы ip-адрес. У меня вот, соответственно, все развернуто на моем docker-е локально. Поэтому здесь только docker. Через двоеточие идет, соответственно, порт. И потом идет название базы данных. Вот как мы здесь видим, у нас база данных называется postgres. Соответственно, нам нужно ее указать, иначе не будет знать ваш драйвер, к чему, собственно говоря, подключаться.

Соответственно, соединение устанавливается сразу при этом. И если такой базы нет или что-то у вас не получилось, он, естественно, вернет ошибку. А в поле conn будет nil. Соответственно, в таком случае, естественно, работать с базой данных мы не можем, что-то мы должны там предпринимать.

После того, как мы получили успешное соединение, мы ставим закрытие соединения. В конце функцию тоже с contex-ом. И поехали в, так сказать, мясистую часть этого. Соответственно, у соединения есть параметр QueryRow. Этот параметр как раз предназначен. Не параметр, простите, а функция, она предназначена для того, чтобы как раз забирать данные из базы данных. Она тоже

использует contex, потому что, в принципе, все функции соединения с базами данных – они, так сказать, trade-oriented, goroutine-oriented. Поэтому они всегда используют какие-то хуки. Для того, чтобы вам из goroutine было проще с ними работать.

Соответственно, после этого идет, так называемый, sql-запрос. То есть все вот эти базы данных – postgres, MySQL, oracle – они все используют какие-то диалекты sql. Я не буду глубоко погружаться в sql. Я просто покажу, как он примерно выглядит. То есть вот самый простой, наверное, sql-запрос. То есть слово, что мы делаем, потом – что мы извлекаем. Ну в данном случае мы что-то извлекаем, да, select – это выбрать, извлечь информацию. Потом у нас список полей, потом откуда мы это извлекаем и, соответственно, имя таблицы. Можно еще добавить к этому, так называемый фильтрующий, запрос, где мы извлекаем. Можно также лимитировать запросы, например. Могут запросы быть не только select, могут быть insert, если вам нужно что-то вставить. То есть insert into users. Потом у нас есть, значит, здесь список полей. То есть в нашем случае будет id, name, rank, например. Соответственно, потом будет values. И у нас какие-то там типы у нас будут здесь значения. Вот.

То есть такие основные запросы. Ну и есть еще запрос update, то есть здесь мы уже как раз обязаны, наверное, поставить where. То есть это рекомендуется, да. И здесь мы уже можем что-то тут установить. Здесь мы можем указать, что для пользователя с id=2, мы, соответственно, устанавливаем какое-то имя. Можно, конечно, без where. Тогда имена по всей таблице у нас проставятся одинаковые. То есть, когда нет where, update делает на все записи в таблице. Но обычно в реально жизни это не используется. Это такая была база sql.

Давайте посмотрим, как это имплементируется уже, непосредственно, в драйвере. В драйвере, как мы видим, sql пишется просто строкой. То есть вы запрос ваш

пишете также, как я его сейчас в консольке написал. Это удобно, потому что можно прямо сразу перенести и скопировать. Но id у нас равен $1. Что это такое? Это (НРЗ 10:51).

То есть у нас здесь после, значит, запроса идет (НРЗ 10:57). Что такое args – это, соответственно, аргументы запроса, как можно догадаться. Но чтобы запросу понять, куда их вставлять, вы должны использовать вот такие вот (НРЗ 11:06) - $1, $2, $3 и так далее. Удобно их использовать в том смысле, что, например, строковый запрос он сам проэкранирует. То есть если какой-то злоумышленник у вас захочет заинжектировать что-то в sql, добавить свой запрос, экранируя ваш оригинальный, у него ничего не получится. Потому что запрос будет проэкранирован.

Ну и в целом удобно, когда в запросах нет реальных данных. Вы можете где-то запрос отдельно подготовить, и потом его сюда вставить. И после этого идут аргументы. Соответственно, QueryRow возвращает, так называемый, Row. Это Wrapper, то есть он, соответственно, содержит в себе сырые данные каким-то образом. И имеет метод scan. Метод scan позволяет вам просто по порядку сосканировать переменные из запроса, то есть те, что база данных вернула, в реальные переменные в вашем коде. В данном случае создал три переменные. Вот эта звездочка значит, что мы выбираем все поля.

И я знаю, что эти поля идут в порядке (НРЗ 12:11). То есть здесь аналогично вот такому вот запросу. И поскольку я знаю их порядок, я могу их смапить. То есть в моем случае rank называется position. И я маплю 1 в id, и в name rank в position. Соответственно, если что-то не получилось, вернется ошибка при скане, при мапинге. Ну и тогда нам тоже здесь, в принципе, делать нечего.

Давайте запустим, посмотрим, как оно работает. Вот видите, у меня тут вывелся name и position one VP Business. Что будет, если я уберу, например, одно поле. Что нам скажет драйвер? Драйвер нам скажет, что количество полей, которое ему удалось получить, не совпадает с количеством полей в скане. Вот как раз вот эта вот ошибка у нас сработает. То есть следите за тем, чтобы у вас в скане было одинаковое количество полей.

Хорошо, мы забрали один столбец с помощью драйвера. Увидели, как это работает. Тут работает метод scan. Давайте теперь посмотрим, как забрать несколько столбцов и как, соответственно, забираются столбцы в объекты. То есть как мапить это в объекты.

Соответственно, здесь, в принципе, у нас тоже самое. То есть здесь у нас была просто инициализация. Здесь я инициализирую contex отдельно, чтобы не таскать ссылки туда-сюда. Ну и дальше создаю, так называемое, pool соединение. В принципе, мы работаем с ним примерно также, как и с обычным соединением. В чем разница pool соединений. Когда вы соединяетесь с базой данных, у вас на вашем компьютере создается connect к базе данных. То есть вот это ваш компьютер, K – компьютер, вот это ваша база данных. D – данных база. Соответственно, если это соединение одно, то после его закрытия, у вас соединения больше нет. То есть образуется такая священная пустота. Каждый раз, когда вам нужно будет новое соединение, вы его заново открываете. Если база данных по какой-то причине разорвало соединение, вам тоже его нужно переоткрыть. Причем ваше текущее соединение еще и может стать невалидным. То есть вы будете думать, что оно открыто. А вместо этого получите ошибочку при запросе.

Поэтому умные дяди придумали делать соединений сразу несколько открывать, заранее, оптом. И их использовать. Это, в принципе, рекомендуемый подход при

работе с реальной базой данных где-то в production-е, потому что, если у вас вдруг что-то отвалилось – ничего страшного, соединение поднимутся. Внутри pgxpool-а есть еще и механизм, который позволяет создать соединения, если их не хватает. До определенного момента, да? Ну и соответственно, у нас здесь намного надежнее, потому что канал у нас не один, а каналов у нас там несколько. Ну вот тут я больше не могу нарисовать, к сожалению.

Ну ладно. Хрен бы с ним. Здесь мы подключились к pool-у, создали структуру для пользователя. Здесь все старые знакомые – поля id, name, rank определенных типов. Сказали, что pool мы пока в общем закрывать не будем, закроем в конце функции. И дальше я использую трюк, я использую библиотеку pgxscan, которая как раз является оберткой над драйвером pgx. И позволяет нам делать более простые запросы. В данном случае мне нужно забрать несколько пользователей. Вот как мы видим здесь у нас полный вывод пользователей, 10 строчек. И есть простой запрос select. Я туда передаю наш slice, передаю драйвер или pool, или одно соединение от драйвера базы данных, и передаю запрос.

Следовательно, что происходит под капотом? Под капотом выполняется наш метод Query. Я изначально использую метод QueryRow для того, чтобы один столбец вывести. Соответственно, метод Query у нас выведет несколько столбцов. Его значение возвращаемой – это, так называемый, rows. И rows – это интерфейс, у которого есть, так называемый, итератор.

В чем прикол? Прикол в том, что базы данных можно эксплуатировать по-разному, и иногда в базе данных очень много полей. То есть там у вас десятки миллионов полей. И естественно, вы эти поля можете забирать по одному, а можете забирать так называемым, курсором. То есть курсов – это специальная структура данных, которая в себе не содержит сразу все поля, она как бы по ним идет

последовательно. То же самое, когда вы читаете, например, большой файл. Вот у нас похожее было нечто работы с файлами.

И поскольку не хочется делать много разных типов, то вот этот вот столбцы, которые мы возвращаем с базы данных – они все организованы подобно курсорам. То есть в данном случае у нас есть метод next, который заберет столбец. То есть условно у нас выглядит это так. Мы забрали, ага, я здесь не могу редактировать, да, мы забрали из базы данных первый столбец. Потом мы с помощью метода… Забрали первый столбец, вызвали метод next, забрали второй столбец, вызвали метод next, забрали третий столбец. Вызвали метод next. Предположим, что столбцов дальше у нас нет. Метод next вернет false.

И мы будем тогда знать, что вот мы три столбца забрали, больше там в этих итераторах ничего нет. И в принципе, метод select после, соответственно, вызова Query – он это, в принципе, и делает scan all. То есть он берет и идет по столбцам. И соответственно, сканирует их уже в назначение. То есть мы передаем ему какой-то интерфейс, куда сосканировать. И соответственно, он его туда сканирует.

То есть здесь какая фишка? Мы передали ему slice. Он знает, что там несколько столбцов и попробует их все засунуть в slice. Соответственно, у метода select тоже есть error. Его тоже можно обработать, он там вернет что-то, если у нас неправильные поля. И соответственно, мы здесь уже, как видите, не мапим их какие-то примитивные типы. Мы уже создали свою структуру. И в принципе, для этого библиотечка и нужна – обернуть вот эти все моменты в структуры.

То есть мы сами не должны брать rows, по ним бежать, потом каждый раз создавать новую структуру, в нее что-то запихивать, запихивать ее в slice. Уже все сделано за нас. Мы передаем ссылку на slice и погнали.

Ну то же самое, соответственно, для одного параметра, для одной переменной, для одной строки из базы данных с параметрами. И здесь все также, как в методе QueryRow. То есть мы просто получаем этот метод. В принципе, я рекомендую всегда использовать такие Wrapper-ы, когда вы работаете с какими-то sql драйверами. Если драйвер не умеет мапить напрямую структуру. Потому что вот каждый раз писать, что for rows next, там значит user, потом открывать кавычки, фигурные скобочки, туда что-то запихивать – это в общем утомляет. А тут так все просто, чик-пык, соответственно, два вызова. И все работает.

Давайте посмотрим, как это работает. Соответственно, здесь можно видеть, здесь можно видеть результат выполнения в этой строке. У нас тут массив со всеми людьми. И соответственно, здесь у нас одна переменная, полноценная с пользователем.

Я долго втирал за первичные ключи, за индексы и за все вот это вот такое, связь таблиц. Давайте посмотрим, как она реально работает. Естественно, связь таблиц тоже обеспечивается путем sql-запроса, sql, так называемого, join запроса. Соответственно, я буду использовать (HP3 20:32) запрос, где у нас все поля, имеющие какую-то отсылку к id, будут соединяться с такими же полями, имеющими отсылку к id. Давайте посмотрим, как это работает непосредственно в самой базе данных, то есть напрямую в запросе, не в Golang-е.

Вот, соответственно, здесь смотрим. Здесь у нас есть уже две таблицы. Можно назвать их как угодно, то есть можно alias-ы им придумать. Вот в данном случае у меня cars называется C, вместо cars, так просто быстрее писать. Я беру имена пользователей, значит, их ранги. Беру бренд, цвет и номер машины. И соединяю их по id-шнику, id пользователя, который равен user_id в машинах. Вот мы видим здесь, это foreign key, то есть это ключ, указывающий на таблицу users.

Что произойдет в этот момент? В таблице users у меня выберутся все колонки, и потом сравнятся, так сказать, склеятся вместе с колонками автомобилей, где совпадает users_id с id пользователя. То есть в данном случае, если у нас в таблице users был один пользователь, допустим John K. И у него была одна запись, то в случае соединения таблиц, у него будет несколько записей, потому что машин у него тоже несколько, как можно увидеть. Здесь, соответственно, несколько машин у него. Вот один, один, один. То есть для соединения таблиц здесь создастся четыре записи с одинаковыми данными.

Но это, в принципе, именно то, что нам нужно. Давайте посмотрим, как это выглядит. Мы видим, что у нас есть Betty P., и у нее четыре автомобиля. У нас есть Jimmy C., у которого два автомобиля, и так далее. То есть эта информация дублирующаяся, но это такой нативный, так сказать, способ для того, чтобы соединять две таблицы в sql-ориентированных языках.

В Golang это выглядит точно также, в принципе. То есть мы здесь пишем запрос. Я здесь добавлю where, чтобы отфильтровать, скажем, все, показать машины и должность для всех (НРЗ 22:57-22:59). И создал структуру. Структура у нас, соответственно, повторяет то, что я забираю в полях. Имя, rank, brand, colour и так далее. Ну то есть это обычный реальный запрос. И ничем он особо не отличается. Здесь я опять же создаю массив, забираю автомобили вместе с пользователями.

И можно видеть, что у нас John K., John K., John K., John K. Ничего удивительного. У нас один только John K. И он, так сказать, генеральный директор с четырьмя автомобилями, которые зарегистрированы в этой компании. Вот, как-то так соединяются таблицы, как-то так появляются сложные структуры.

Ну ладно, мы с базы все забирали, забирали. Давайте что-нибудь туда вставим. Со вставкой все вообще супер просто. Я здесь не использую никаких оберток,

потому что pgxscan, он предназначен для сканирования, для всяких select-запросов. Потому что вам нужно их смапить каким-то образом, нужно их обрабатывать. Вот это вот все. Нужно как-то с ними взаимодействовать.

В случае со вставкой все очень просто. Тот запрос, который я уже показывал, insert into. Мы здесь добавим новую машину. Соответственно, здесь просто куча мапперов, просто куча (НРЗ 24:13) и куча, соответственно, переменных, которые туда мапятся. Если вы туда объект засовываете, то тут, к сожалению, придется перемапливать поля. Способа простого засунуть объект драйвер у нас не предлагает. Но, в принципе, после этого exec, мы можем, соответственно, получить command tag, то есть получить результат выполнения функции. То, что нам postgres напишет. И соответственно, ошибку. Но ошибка у нас возникнет, только если что-то не удалось сделать.

Соответственно, как-то так. Вызывается эта функция и добавляется туда автомобиль. Соответственно, вот мы видим, что ошибки у нас нет. И если мы сейчас пойдем в машины, мы увидим, что у нас добавилась запись 21, красный Peugeot. Вот, как-то так.