Давайте соединим наши знания воедино из всех предыдущих частей и соберем наш HTTP сервер для решения задачи прикладной. Соответственно, задача в чём состоит? У нас есть какой-то офис, в офисе работают какие-то люди, у этих людей есть машины и есть, значит, подземная площадка под офисом, где все паркуются. Соответственно, охране этой подземной площадки, вернее, автоматической системе с шлагбаумом нужно знать: цвет, марку и номер автомобиля для того, чтобы его пропустить. И, соответственно, руководство, которое ставило задачу, подумало, что ещё не плохо было бы хранить связь пользователя с машиной чтобы знать, кому эта машина принадлежит.

Мы получили техническое задание на сервис, включающее в себя 6 методов: создать пользователя…, создать машину в связке с пользователем, получить список пользователей, найти пользователя по его идентификатору. То же самое с получением списка машин и получить машину по идентификатору. Машины всегда связаны с пользователем, у машины всегда есть цвет, какая-то марка и, соответственно, номер. Давайте посмотрим на имплементацию этого сервера и разберемся, как эта задача была решена.

Давайте посмотрим на структуру проекта. Структуру проекта я старался сделать близкой к, соответственно, уже каким-то production-решениям, но не потому, что это production, не потому что так надо делать, а потому что это достаточно удобно поддерживать и сразу понятно: что где лежит. Давайте разберемся.

Соответственно, вот корень проекта: у нас есть здесь go.mod, понятно, что это такое? Описание go модуля со всеми зависимостями, есть docker-compose, где лежит моя тестовая база данных для моего личного тестирования – это хороший вообще, в принципе, тон класть тестовые вещи уже в проект. Потому что на унесении в production это, естественно, сети в compose-вские файлы использоваться не будут, но разработчик, когда pool-ит архив, у него сразу всё есть

для работы. Соответственно, папочка migrations с sql-ными миграциями, мы используем postgres для этой задачи, соответственно миграции у нас тоже postgre-совские. Если у вас какая-то есть другая база данных – у вас будут другие миграции. Здесь мы должны отделить наши alter sequence-ы. То есть всё понятно: при старте проекта мы просто выполняем миграции.

Папочка internals – мы к ней вернёмся чуть позже. Папочка cmd содержит в себе все команды, которые запускаются из консоли. У нас здесь есть только команда main, которая запустит наш сервер.

Папочка api, которая в себе содержит наши HTTP-шные маршруты, то есть наши (НРЗ 03:05), к которым мы будем обращаться, те, которые у нас здесь есть – эти все ресурсики HTTP-шные. И содержится middleware – тот, про который я говорил, то есть какую-то (НРЗ 03:23), которая привязывается к середине запроса и что-то с этим запросом может сделать.

Папочка Internals содержит в себе папочку cfg – это наша конфигурация, мы обсуждали уже флаги и переменные окружения, посмотрим? как это работает чуть позже.

Папочка app содержит в себе main, иногда его называют app, то есть точку входа именно в наш сервер. То есть это cmd запустит наш сервер, но сама логика сервера находится именно в internals app, то есть internals – это то, что принадлежит нашему HTTP проекту. Папочка db, которая отвечает за работу…, в ней всё, что отвечает за работу с базой данных. Handlers – это всё, что…, те самые handler-ы, про которые я говорил, те контролеры в других терминологиях, которые уже непосредственно отвечают за обработку этих HTTP запросов. Наши модельки, которые используются для получения данных в базы, оборачивание их

в гос. структуры. И наши процессоры, которые делают какую-то логику. В других терминологиях они могут ещё называться «сервисами».

В чём удобство здесь? – Здесь удобство в том, что основные логические блоки уже разделены. Если вам нужно, например, добавить новую модель – вы идете в internals – app – models, добавляете новую модель. Если вам нужно добавить какой-то номер, маршрут – вы идете в api – routes. Если вы генерируете документацию, причём папочка api отдельно, хотя, казалось бы, она принадлежит приложению, это всё-таки часть приложения. Потому что, если вы вдруг захотите сгенерировать так называемую api-документацию, то есть документ, в котором у вас описаны все ваши маршруты и как с ними взаимодействовать, вы можете написать документацию прям в ваших route-ах и сгенерировать уже отдельно api документацию, и как-то ещё положить куда-то. То есть вам не нужно указывать специфически будет конкретно папочку в internals, у вас всё лежит в api.

Теперь уже давайте заглянем в наш файлик go.mod и посмотрим, какие модули мы используем для этого проекта. Модули мы используем, в принципе, те же самые, которые используются на больших проектах. Обычно модули в Golang-е выбираются по принципу списочка, который называется awesome go, на GitHub-е он существует, значит, curated list, там есть куча всего. То есть по вообще всем направлениям, которые возможны. Обычно там собраны действительно, хорошие вещи, в которых есть какие-то звездочки, какой-то recognition. И если вы используете что-то в вашем проекте – рекомендую вообще сюда, в принципе, заглянуть, – это довольно-таки офигенная штука.

Мы используем то, что используют вообще, в принципе, наши коллеги-developer-ы, то есть те, которые пишут уже непосредственно сервисы. Начинаем мы с уже известных нам pgx и scany, то есть коннектора к базе данных Postgres и scany того самого wrapper-а, который облегчает нам извлечение данных из базы данных.

Соответственно мне что нравится особенно в модулях go – они всё содержат в себе ссылки на GitHub, поэтому можем их просто открыть здесь, посмотреть. Опять же, посмотреть на звездочки, посмотреть на описание, то же самое с этим драйвером Postgres-овским, – тоже можем посмотреть на описание, например, сразу просто не отходя от кассы.

Следующая вещь, которую вы ещё не видели – это mux, то есть мы в прошлых примерах при разборке HTTP показывали, что эти, значит, пути HTTP-шные – их можно привязывать к конкретным функциям, которые будут их обрабатывать с помощью стандартных методов (HP3 07:36) в HTTP handler. И всё хорошо, handler работает, всё здорово, но, например, если у вас есть какая-то…, какой-то метод, который поддерживает только post, не поддерживает get, например. То есть у вас есть…, только можно в пользователях, например, с созданием, можно только создать пользователя с помощью post-метода, а get должен ошибку вернуть. Вы должны это руками сделать: вы должны проверить if, request, точка метод равно get или равно post, значит, мы делаем else – мы возвращаем ошибку, её как-то обрабатывать должны, запихнуть… Не очень удобно. И для этого существуют…, в том числе для этого существует такой роутер gorilla/mux, опять же, очень популярный проект, как можно видеть.

В чём удобство? Он, во-первых, сам по себе довольно-таки, удобный. То есть он обладает простым интерфейсом. У него больше гораздо методов matching-а, то есть он может и по префиксам match-ить, то есть не только по строгому соответствию строки. И по методам, и по схемам, header-ы может передавать и так далее, можно свои функции определить. Короче, он довольно-таки удобный и довольно-таки, расширяемый. Из-за этого, он практически, повсеместно используется в различных web-проектах. И мы тут не будем исключением: возьмём mux. Mux работает со стандартным пакетом NAT HTTP и просто вполне спокойно в качестве handler-а в этот пакет встаёт. Дальше.

Дальше у нас существуют logrus. Logrus – это, соответственно, logger. В чём фишка? Любой ваш проект, который вы делаете, должен в себе содержать какую-то отладочную информацию, потому что, если что-то вдруг сломалось – удачи понять это без log-ов. То есть надо его разбирать, пытаться воспроизвести эту ошибку. Как её воспроизвести если вы ничего о ней не знаете, потому что ваш проект ничего не пишет? В общем, про log-и я долго распинаться не буду. Тут всё понятно.

Соответственно, существуют некоторое количество logger-ов, потому что стандартный fmt.print – он просто печатает строку в консоль, а вам нужно, например, иметь возможность её в файл сложить или её как-то отформатировать, или как-то её обработать. Опять же, ввести всякие разные уровни, log-ирование – то есть какие-то log-и важны, какие-то – не важны. Опять же, всем этим гибко управлять. Можно самому, конечно, написать, но есть уже готовые решения, например, этот logrus, который как раз всё делает. Он и форматирует, он умеет и разные уровни делать, и легко встраивается таким образом log., log… Где пример у него? Господи… log.info – здесь мы можем написать log.WithFields, отформатировать что-то. В общем, у него куча настроек, он очень удобный, и я его тоже буду применять в своём проекте.

И последнее, что здесь есть – это viper. Viper – это такая повсеместная тоже используемая штука для чтения всяких конфигураций. То есть ваш проект будет содержать какую-то конфигурацию, будет флаги содержать, будет содержать переменное окружение, как мы обсудили. Возможно, какие-то текстовые файлы вы туда будете выгружать с какой-то информацией для того, как проекту работать.

Viper умеет это всё читать, сводить воедино в интерфейс, подставлять какие-то значения по умолчанию в том случае, если у вас нет этого в переменных окружениях, или (НРЗ 11:24) читать json-ы, (НРЗ 11:25), INI-config-и. В общем, у

него…, даже Java properties он может, хотя ни разу не видел, чтоб кто-то это использовал. Соответственно, очень удобная штука, тоже достаточно production ready – её много кто использует, и мы тоже вполне себе спокойненько заюзаем.

Чего здесь нет? На самом деле, кто-нибудь уже мог…, кто уже с Go знаком, тот мог обратить внимание, что: «Ага, fasthttp не подключили». Не подключили по простой причине: fasthttp – это специализированная (НРЗ 12:07), она может вам вполне пригодиться в DevOps-овской жизни, которая специально оптимизирована для отдачи огромная числа маленьких запросов. Например, если у вас есть какой-то track-ер, если у вас есть какая-то life прога, которая дергает постоянно клиента. Если у вас есть какой-то endpoint, который торчит в интернет и его надо прям…, его постоянно кто-нибудь опрашивает, то fasthttp – он прям, сэкономит вам кучу ресурсов, и он специально оптимизирован для таких вещей. Тут написано: для того, чтобы обрабатывать тысячи маленьких или средних запросов в секунду.

Соответственно, fasthttp для этого пожертвовал многим…, относительно многим, например, поддержкой http 2 web socket-ов то есть какими-то постоянными соединениями. У него очень простая реализация…, довольно простая реализация с минимумом движущихся частей, поэтому мы его использовать в данном примере не будем. Наш сервис с нашими машинками и гаражами – не настолько требователен к запросам, и вряд ли его кто-то будет опрашивать 1000 раз в секунду. Машины так быстро не ездят, поэтому в этом довольно мало смысла. Если вы хотите его использовать – он работает схожим образом с net/http, но handler-ы там отличаются. Поэтому здесь уже будет несколько сложнее, но тем не менее, переехать вполне возможно, если вдруг вам это будет нужно. Поехали дальше.

Наши данные надо где-то хранить, поэтому нам нужна база данных. Я уже показывал файлик с миграциями так мельком. Давайте посмотрим, что эти

миграции делают. Я опять же использую клиент, встроенный в мою (НРЗ 14:02) – очень удобный клиент для баз данных реляционных. Соответственно, у меня есть всего 2 таблицы, то есть мне надо хранить пользователей и машины – всего 2 сущности. И машины хранят все ссылки на пользователей.

Это, в принципе, всё ТЗ, которое у меня есть, всё остальное – да только (НРЗ 14:21). То есть пользователь у нас имеет идентификатор, имеет имя и его позицию в компании – она же «ранг». Выглядит это следующим образом: это имя, это его ранг, то есть ничего особенного. Соответственно, машины, машины несколько сложнее, но как сложнее? Опять же, у них 3 поля: цвет, бренд и номер. Цвет у нас просто задан буквами, иногда для экономии ресурсов и для того, чтобы быстрее работало, цвет задаётся цифрами. И в приложении просто пишется, что голубой – это у нас 1, белый – это 2, либо делается RGB так называемая палитра, где цвет кодируется тремя битами и это полное значение…, 3-мя байтами, простите, полное значение кладется уже непосредственно в таблицу. Я не стал так извращаться, опять же у нас там небольшая база, достаточно каких-то текстовых значений, то же самое с brand-ом, то же самое с license plate, то же самое с регистрационным номером.

Соответственно, есть у него ключ, указывающий на id-шник пользователей, то есть мы здесь видим: даже нарисовано в нашей схемке, что у нас user_id – это поле, указывает на user’s id – то есть на id-шник этих пользователей. И по-хорошему, нам этого достаточно на данный момент для того, чтобы имплементировать нашу систему.

И время разобраться, что у нас происходит в наших файлах. Начнём мы с папочки cmd, с команды main.go – это наша точка входа, это то, что у нас будет запускаться, то что запускается наш web-server. И что мы здесь видим? У нас есть функция main в package main, то есть это наша точка входа. Соответственно,

первое что мы видим – у нас загружается конфигурация, посмотрим на это чуть позже. Cfg – это моя история: это модуль, который… То есть, соответственно, package, который я написал и метод, который тоже я написал.

Дальше мы делаем context, здесь context с отменой и слушаем сигналы операционной системы. То есть наш сервер должен работать следующим образом: мы его подняли, открыли соединение в базе данных, подняли всякие ресурсы, подготовили url и начали ждать. Оккупировали, естественно, порт и начали ждать. Если система хочет наш процесс завершить по какой-то причине – мы должны на это отреагировать: закрыть соединение к базе данных, чтоб база данных не думала, что мы там всё ещё висим. Закрыть…, сняться с port-а, всё правильно погасить и сделать так называемый, graceful shutdown, то есть правильное…, правильный выход из приложения, закрыть все файловые дескрипторы, закрыть соединения. В общем, не оставить после себя никаких следов в системе.

Поэтому мы здесь создаем наш канал с os.Signal и используем пакет signal, входящий в пакет os для того, чтобы поймать как раз сигнал операционной системы о том, что у нас завершилось наше приложение. После этого мы создаём наш сервер – это опять же app, это мой package, который я сделал. Создаём наш сервер и в функции, которая привязана к нашему каналу, мы ловим исключения…, ловим, вернее, наш сигнал и завершаемся по этому сигналу и заодно, отменяем все контексты, которые мы используем для того, чтобы поднять базу данных и прочее. После этого мы уже готовы, мы можем начинать служить нашему серверу.

Давайте посмотрим, что происходит глубже. И первое, что мы видим, первая строчка – это загрузка конфигурации, которая возвращает в себя объект config, используемый далее. Давайте посмотрим, что здесь происходит.

Здесь я как раз использую viper, то есть нашего чтеца конфигурации, создаю его instance через New. И у меня здесь в проекте есть только конфигурация из environment-a – из переменных окружения. Для того чтобы не путать свои переменные окружения с другими переменными окружения, системными, например, у меня у моего приложения будет префикс. Это, в принципе, рекомендуемая история для всех переменных окружения. То есть вы там пишете…, у вас есть приложение, которое делает интернет-магазин – оно называется Amazon, Ozon. И ваш префикс для переменных окружения должен быть Ozon подчеркивание App, что-нибудь, потому что в таком случае у вас будет… У вас не будет в системе коллизии, то есть у вас не будет такого, что ваше приложение считало переменную окружения чего-то вообще другого, поэтому называть просто переменную host, port и так далее – это не очень круто. Не всегда окружение изолировано, не всегда только те переменные, которые вы хотите. Как правило да, с Kubernetes-ом и Docker-ом это так, но я люблю перестраховываться.

Опять же, это можно задать, например, в файлике конфигурации – это название префикса, его тоже грузить. Я оставлю префикс для переменных окружений. Дальше, всё что я читаю – это…, эти переменные будут читаться…, эти переменные должны быть заданы в системе каким-то таким образом. SERV_PORT, SERV_DBUSER и так далее. Здесь я ставлю значение по умолчанию, если port не задан, если его нет в переменных окружения – у меня «8080», естественно, USER для базы данных не задан, у меня пользователь «postgres». Если нет пароля – опять же, host базы данных я тоже не задаю, хотя мог бы, например, local host, но для чистоты эксперимента мы не будем этого делать. Соответственно, порт базы данных, имя, здесь мы вгружаем переменное окружение и здесь мы собираем мою структуру. Структура имеет отношение к конфигурации, поэтому есть её смысл хранить в package cfg, структура тоже называется cfg – она в себе содержит то, что мы тут вгружаем.

И чем удобно viper: имеет в себе функцию Unmarshal, то есть он сличает переменное окружение с…, название переменных окружений с полями структуры и их просто туда записывает. И опять же, у структурки моей есть метод GetDBString. Его можно было бы вынести отдельно, но это однострочник, опять же файлик маленький, поэтому в этом нет никакого…, никакой проблемы здесь её указать. Она, естественно, вернет строку подключения для нашей конфигурации, для нашего конфигуратора баз данных, для нашего коннектора к базам данных. Как-то так. Соответственно, после получения переменной окружения эти…, это переменное окружение идёт у нас непосредственно в наш сервер.

Итак, следующее, что мы видим – это создание нашего сервера непосредственно, и 2 метода: serve и shutdown. Наш сервер – это структура с методами, это её конструкторы. Таким образом, в Go создаются конструкторы наших структур.

В конструктор мы передаем те значения, без которых наш сервер…, наш класс работать не может. Наш сервер, в данном случае, не может работать без конфигурации, без контекста извне. Здесь мы создаём наш сервер, его возвращаем и над ним ещё можно делать некоторые операции, то есть его стартовать и, соответственно, его выключать. Дальше мы тут делаем log-ирование о том, что наш сервер стартовал. Нам интересно это будет знать, что сервер наш стартовал, работает в log-ах. И далее мы, например, рецензируем pool соединения в базе данных также, как мы это делали в нашей предыдущей лекции: соединение с базой данных. Напоминаю, что pool – это несколько соединений в базе данных, в какой-то момент времени может работать одно или несколько соединений. Остальные просто висят и ждут своего часа и таким образом, вы не теряете ресурсы на переоткрытие соединений.

Если на этом момент что-то пошло не так, то мы здесь делаем fatal, то есть Print – это просто информация для нас, fatal сделает ещё os.Exit, то есть завершит наш

сервис. Потому что, если мы не можем подключиться к нашей базе данных, у нас отсутствует основной механизм для работы с нашим приложением, поэтому дальше смысла работать нет.

И дальше создаем мы здесь instance для нашей базы данных, требующей начало сервера. Обращаю внимание, что здесь всё делается через конструкторов. Это удобно, потому что вы сразу видите, какие зависимости есть. У вас они инициализируются не где-то под капотом, внутри, а вы видите снаружи, что: «Ага, этот сервис использует базу данных. Этот сервис использует конфигурацию», и так далее. Дальше наши processor-ы и наши handler-ы. Тут в ложной зависимости processor-ы используют наши storage-ы, нашу логику работы с базами данных, handler-ы используют processor-ы.

Дальше мы создаем route-ы из пакета api, подгружаем наш middleware и стартуем сервер через http.server. Опять же, у нас в предыдущих лекциях был http.server, мы там его стартовали через http.ListenAndServe. В этом случае создается стандартный какой-то server. В нашем случае у нас он будет разбит на несколько модулей, то есть там нужно уметь работать с этим сервером. И мы его поэтому помещаем в специальное поле в нашем сервере, в нашей структуре. Здесь мы загружаем, опять же, из конфигурации наш Port, и передаем в handler наш router – наш mux, то есть то, о чём я говорил, когда говорил, что mux нативно работает с HTTP. И после этого мы можем смело сказать, что наш сервис стартанул.

На самом деле, больше… Вернее не так. На самом деле, после… Этот метод – он зависнет в потоке, то есть он будет постоянно слушать и что-то serve-ить, поэтому этот log вы не увидите. И, значит, я подразумеваю, что после создания сервера у нас всё хорошо, нормально, и наш сервер стартанул. Если у нас что-то пошло ненормально, я это тоже увижу в log-ах.

И функция Shutdown, которая к нам…, нам выполнится, если нам приходит от системы уведомление, что пора бы закончить всё. Всё выключает, закрывает соединение и закрывает (НРЗ 25:33) соединения к базе данных. В конце вызывает cancel из контекста. Выключает сам сервер и после этого печатает, что сервер вышел и не реагирует на ошибку. В этом, в общем-то, весь наш сервер, и всё самое интересное будет теперь происходить под капотом.

Итак, значит, мы разобрались с нашим основным сервером, с нашими командами и обращаю внимание, что всё, в принципе, поделено по принципу ответственности. То есть здесь в package у нас происходит изначальный сборка server-а и выключение server-ов, включение-выключение server-а, то есть управление глобальным состоянием системы. Здесь у нас происходит сборка всех зависимостей для server-а, то есть здесь создаются все классы, с которыми наш server будет работать. Они же уничтожаются здесь же, то есть это одна функция. И, в принципе, дальше мы продолжим делать то же самое, то есть делать package-ы и структуры методов этих package-й, которые делают что-то одно. И яркий пример – это создание путей.

Пути у нас лежат, как я уже говорил, в api. И пути у нас отвечают за создание пути. Они получают все handler-ы, создавать handler-ы – это не их задача. Создавать handler-ы должен кто-то ещё. В нашем случае – server. Они их просто получают и возвращают router, который знает о путях. Мы создаём этот router с помощью mux и туда вписываем значение наших ресурсов. Обращаю внимание, что мы после передачи значений и определения функции для обработки этого ресурса HTTP, указываем ещё и метод. То есть в нашем случае «users/create» принимает только post, «users/list» принимает только «GET». Опять же, методы можно через запятую указывать, вы можете здесь написать «GET» и «POST», и так далее. Если вы ничего не укажете – будет любой метод приниматься.

И, соответственно, метод find в себе содержит id. Id – это специальное значение, можно увидеть, что в методе Find оно будет передаваться в Vars, мы ещё разберемся с handler-ами. И оно в себе содержит только цифры. Если оно в себе будет содержать что-то другое – опять же, mug сругнется, не сможет его найти и вернет нам ошибку.

Здесь у нас методы для машин и здесь у нас методы для пользователей. Всё собрано в одном месте. Этот класс отвечает только за создание путей и в нашем случае ещё отвечает за обработку ошибок. То есть существует некоторое количество ошибок, которые уже связаны непосредственно с route-ингом, например, если у нас нет такого пути или id-шник, например, stock-овый, а не цифровой. Это значит, что пользователь забрел куда-то не туда и ему надо рассказать, куда ему идти либо просто сказать, что это неправильный путь. Соответственно, эти ошибки порождаются самим router-ом, то есть мы уже внутри сервера за них никак не отвечаем, потому что наши handler-ы работают только в случае, если пользователь ввел правильный адрес здесь. И поэтому router сам возвращает какие-то ошибки, и они нас не очень устраивают, я потом покажу – почему. И, соответственно, мы здесь определяем отдельный handler, который будет отвечать именно за ошибки NotFound. То есть за ошибки, когда маршрут не совпадает с заявленным, мы тоже имеем над этим контроль.

И давайте посмотрим уже теперь на сами handler-ы, то есть начнём с самого простого: UsersHandler – это у нас handler, отвечающий за вывод одного пользователя, списка пользователей и создание пользователя. Соответственно, новый handler: здесь создается просто handler, в него передается processor, processor – это наш сервис, работающий уже непосредственно с данными, каким-то образом их обрабатывающий. И у handler-а, опять же, исходим из задач: у handler-а есть одна задача – это принять запрос, мы уже знакомы с этим и это видели в HTTP, в нашем сервере клиенте…, принять запрос. Его, соответственно,

декодировать, то есть получить список параметров этого запроса и передать их уже дальше в processor. При возврате каких-то данных из processor-а, их обернуть в нужный нам формат и отправить обратно.

В качестве базового протокола обмена мы используем протокол json, и поэтому у нас здесь существует 2 метода: WrapError и WrapOK. Эти 2 метода я некоторым образом наследую, то есть в том же package handlers я объявляю методы WrapErrorWithStatus и WrapOK. И здесь я возвращаю json, и, соответственно, даю знать нашему конечному потребителю api, что контент type у нас json. Вместе с этим возвращаю статус. Напоминаю, что ResponseWriter – это то, что нам нужно изменить. То есть мы тут его получаем в наш метод – нам нужно его изменить, и таким образом, мы пишем ответ. То есть здесь мы получаем наш универсальную map-у, то есть строки и что-то. И её Marshal-изируем в json, и её возвращаем со статусом OK.

В случае ошибки мы берем текстовое описание ошибки, и у нас тоже есть единый формат: «result» и «data». Тоже он в json-е, и здесь тоже мы его возвращаем таким образом.

Также есть метод для упрощения работы. То есть по умолчанию в моём мире, в мире моего сервиса если вы что-то неправильно сделали в запросе – вы получите ошибку 400: «плохой запрос». Если у вас, например, не ошибка о том, что router не может найти путь, тогда ошибка должна быть 404 по стандарту, поэтому здесь 2 метода: 1 метод возвращает этот BadRequest всегда с оборачиванием ошибки. И 2-й метод возвращает что угодно, и вы можете сами ему определить ошибку.

Соответственно, наш handler имеет 3 метода: найти пользователя, список пользователей и создать пользователя. Самый простой метод – это список пользователя, здесь просто получаем набор параметров из запроса. Я сейчас

откроюкак раз список пользователей. Вот, смотрите, это GET запрос, здесь у нас путь, который мы уже видели, а здесь имя. Имя – это, в данном случае, параметр. Внутри HTTP он развернется как Values – это map-а из строк и массива строк, потому что вы можете, опять же, в HTTP, например, сделать вот так и получить здесь массив. Нам, соответственно, это не надо. У нас…, мы знаем, что там будет одно значение. И в соответствии с этим мы можем работать с параметрами этого запроса через метод Get, который нам вернет либо массив, либо 1-ю переменную.

Так, следовательно, здесь что ещё нам нужно знать? Что наши массивы могут быть обернуты в кавычки… Наши параметры могут быть обернуты в кавычки, потому что так делает HTTP сервер. Нам эти кавычки не нужны и это как раз ответственность нашего handler-а – их отрезать, потому что handler отделяет нашу логику, которая ждет фильтра, например, по имени или каких-то параметров от представления этой логики. Что этот фильтр передается у нас в name, что у него такие-то, такие-то строковые значения – это всё делает handler. Если наш процессор вернул какую-либо ошибку с пользователя, мы возвращаем эту ошибку дальше (НРЗ 34:33). Если нет, то мы выдаем данные, выдаем эти параметры и оборачиваем их в наш OK. Выглядит это таким образом: вот наш OK, вот наши параметры, вот наш ответ, обернутый уже в json.

Похожим образом работает метод find за исключением того, что здесь я беру Vars. Vars – это, я напоминаю, то, что у нас приходит по умолчанию в запросе, в таких именованных значениях, и мы можем к нему обратиться по id. Если он, соответственно, пустой, пользователя мы найти сразу не можем и возвращаем ошибку валидации – это тоже ответственность нашего handler-а. Если можем, ответственность нашего handler-а это id разобрать, его отпарсить в Int, потому что id у нас текстовое… строк… Id у нас числовые, и по числам мы и ищем. И таким образом, у нас метод FindUser в processor-е тоже специализирован. То есть это не ответственность processor-а конвертировать id из строки в число.

Соответственно, опять же нет никакой ошибки, возвращаем пользователя, и это будет выглядеть таким образом: «result»: «OK» и какая-то «data» – это нормальный формат. Обычно примерно так и выглядят api-шки, то есть либо у вас есть какая-то data, какой-то result, какая-то сверху обёртка, может быть, какой-то код ещё ошибки для автоматической обработки и что-нибудь в таком духе.

Здесь я пошёл более простым путем, никаких отдельных кодов у меня нет: «result» или «OK» или «error» и что-то в data лежит, или в result... В смысле, в data что-то лежит.

И последний метод: метод Create, здесь мы уже создаем нашу модельку, а на модельку мы посмотрим несколько позже и её декодируем из json-а, который нам придёт в Body. Поскольку Decoder, опять же, очень удобно работать с Reader-ом и Body у нас, как мы разобрались уже, имплементирует Reader, очень удобно с этим работать и всё сразу декодировать. Соответственно, если это валидный json – всё проходит, не валидный – мы опять же должны дать об этом пользователю знать. Потом передаем всё опять же в processor. Processor создаёт пользователя, мы всё оборачиваем и возвращаем OK. Этот метод у нас post, потому что передача json-а в Body – это то, что обычно происходит в методах post или put. И мы должны его раскодировать и для этого лучше использовать метод post. Вы можете теоретически передавать Body в get-е, но как правило, это не является признаком хорошего тона и является нарушением стандарта HTTP.

Как-то так, то есть таким образом работает handler и задача handler-ов – это просто раскодировать то, что у вас пришло. Возможно, вернуть какие-то ошибки сразу при декодировании, если таковые есть. Как-то обработать и вернуть ошибки возможные, которые появились уже при дальнейшем процессинге в формате, понятном пользователю, то есть в формате json.

Опять же, выглядит это примерно так: если я хочу найти несуществующего пользователя – у меня будет здесь «user not found» с result-ом «error». Если я передаю что-то, здесь должна быть правильная обработка, никаких сырых массивов, никаких структур в string-овом виде, то есть за всё отвечает наш handler – он оборачивает все ошибки и все результаты, у нас в нашем случае, в json и больше ничего не делает, всё остальное делается процессорами.

В принципе, 2-й handler, который у нас cars – делает примерно то же самое. Здесь, практически, ничего нет за исключением того, что, например, в списке пользователей у нас есть…, списки машин, у нас есть возможность отсортировать по пользовательскому id. И если вдруг он не числовой – мы его возвращаем. Но и параметров для сортировки, для фильтрации у нас несколько больше: можно отфильтровать по бренду, по цвету и по совпадению с номерами этого автомобиля. А так вы не особо найдете какой-то большой разницы, processor у нас другой, методы называются FindCar… ListCars и так далее.

В общем-то, handler-ы как-то работают таким образом. Интересный наш handler, опять же NotFound – это точно такой же handler, то есть точно такой же, как и все остальные. В него передается ResponseWriter и Request – то, что запрошено и то, на что надо ответить, и мы можем здесь что-то сделать. В моем случае я оборачиваю ошибку и возвращаю то, что статус не найден, потому что иначе… я сейчас покажу это. Например, такого метода у меня точно нет. Возвращаю 404 «not found». Если я не оберну в json, и опять же, не верну json, то здесь вернется текст, а то, что (HP3 40:01) «not found», какая-нибудь стандартная ошибка, и меня это не устраивает. Я хочу, чтоб мой сервер возвращал всегда json-ы. Если вы хотите определить какой-то свой handler в коде, соответственно просто вы определяете функцию, которая принимает в себя ResponseWriter и Request, и каким-то образом с ней работаете – это стандартный формат mux.

И после разборки наших handler-ов мы уже можем сходить в processor и узнать, чем там живёт наш server. Processor точно также создаётся через конструктор, в него передается опять же база данных. Мы уже это всё видели – не очень интересно. И давайте посмотрим уже непосредственно на методы.

То есть метод GetList…, ListUsers практически ничего не делает кроме того, что принимает в себя список пользователей и всегда возвращает в неё в качестве ошибки. Почему? – Потому что, если мы не нашли ни одного пользователя в списке – это не является ошибкой и это, в принципе, практически единственная вещь, которая может с нами произойти кроме отвалившегося connect-а от базы данных. Условно, потому что произойти-то может всё, что угодно, но я здесь не вижу смысла возвращать никакие ошибки именно пользователям. Соответственно, FindUser может быть такой, что пользователь не найден. Мы это проверим, сверив нашего пользователя с пришедшим id-шником. В противном случае возвращаем, опять же, всё. Сличаем модельку пользователя без ошибок.

И в создании пользователя мы проверяем, что имя не пустое, могли вы это проверить естественно в handler-е, но handler не должен знать ничего о нашей логике. То есть для handler-а, если json valid-ный и, например, там пустые поля, но он может это, тем не менее, разобрать нашу модельку пользователя. То есть она содержит Id, Name и Rank, то всё в порядке, с точки зрения handler-а, и вся внутренняя логика…, уже вся бизнес, так называемая, логика сервиса должна содержаться именно в процессоре. Наш процессор не очень нагружен логикой, как можно увидеть. Здесь пара валидаций и вызов непосредственно наших процессоров. То же самое, в принципе…, то есть наших storage-й, то есть сохранение работ непосредственно с базой данных или извлечение. В car_processor немножко всё веселее – мы здесь ещё отдельно обрабатываем в создании машин и ошибки пустого цвета, бренда и прочего, но опять же,

глобально здесь ничего особенно: ничего нового, чего мы не видели – здесь просто логика.

И как правило, в бизнес-приложениях у вас здесь будет всякая различная проверка на какие-то внутренние соответствия, сравнения по шаблонам, различные обработки и пере-процессинги, то здесь ничего такого сверхъестественного твориться не будет, но это важный слой, поскольку именно он определит нашу бизнес-логику. Наша база данных не должна работать с бизнес-логикой как таковой. Наши handler-ы тоже ничего о нашем бизнесе не должны знать. Если нам нужно внести какие-то изменения в логику и работу приложения – она должна вноситься в processor-ах.

И последнее, что у нас есть из наших слоёв нашей HTTP-шной луковицы – это наши storage-ы, наши системы, которые работают уже непосредственно с базовой данных. У них есть назначение подать запрос в базу данных с определенным параметрами, получить оттуда данные, их обернуть в объект и вернуть на уровень выше.

Логика, которую они реализуют – это собирание запроса и его передача в базу данных. Ничего о, например, том, как к ним попали запросы, как к ним попали параметры с базы данных – их не волнует. И мы начнем опять же с UsersStorage – здесь структурка, принимающая в себя Pool соединений – это обычный connection для нас, следовательно, мы можем пользоваться как connection-ом.

Здесь убираем TODO-ху и первый метод – это получить список пользователя. Мы знаем, что у нас здесь есть фильтр по имени. И возвращаем мы оттуда slice модельки пользователя. Показываем модельку пользователя, один name, rank, можно легко заметить, что это удивительным образом совпадает с таблицей пользователей и, следовательно, мы можем спокойненько смапить эти данные,

смапить эти поля в уже непосредственно модель пользователя. Соответственно, здесь мы проверяем – есть ли у нас фильтр по имени. То есть как это работает?

Я напоминаю: мы после запроса ставим WHERE и можно туда передавать какие-то параметры. Параметры у нас, соответственно, нумерованные: $1, $2, $3 и после того, как мы их передали, ещё нам нужно к ним передать значение, то есть если у вас 3 параметра – 3 значения, 2–2 и так далее. Соответственно, здесь у нас только 1 параметр. Мы смотрим, если он не пуст…, если он не пуст – добавляем WHERE. Здесь я решил искать по нестрогому соответствию. Нестрогое соответствие обозначает, что сейчас покажу… Нестрогое соответствие обозначает, что у меня есть пользователи: «John», «Lana», «Jimmy»… И я могу, например, найти их по имени…, могу их найти по имени «J». Соответственно, здесь он возьмёт только часть этого имени и покажет мне всех пользователей с…, у которых есть в имени слово «J». Также могу найти по букве «m».

Здесь понятно всё, что мы в данных получаем нестрогое соответствие. Если у нас здесь было бы имя равно $1, то он бы искал точно по соответствию. Нас это обычно не очень интересует в случае текстовых данных, нам интереснее получить, как правило, нестрогое соответствие. Опять же, нестрогое соответствие реализуется в запросах типа LIKE путём таких запросов. Если мы сейчас этот запрос написали бы в базу данных – это было бы примерно так: WHERE name LIKE `%J%`, и здесь у нас были бы процентики. Процентики обозначают, что мы ищем букву, и перед ней может быть любой символ и после неё, может быть, любой символ. Если б мы искали только так: мы бы искали все имена, начинающиеся с буквы «J». Если б мы искали так, мы бы искали все имена, оканчивающиеся буквой «J». У LIKE-а есть также ILIKE, и он позволяет ещё и искать (НРЗ 47:16), то есть между большой «j» и «J» – он не будет делать разницы. Соответственно, здесь мы добавляем эти самые процентики в фильтр и подставляем их в аргументы.

Опять же, наш хороший знакомый pgxscan, select. Здесь хотелось бы обратить внимание, что аргументы у нас – это массив…, то есть это (НРЗ 47:46), который можно передавать данные через запятую. Мы здесь собираем, ссылаясь, просто так его засунуть мы не можем – это несоответствие типов. То есть он подумает, что это аргумент такой, но мы можем этот slice развернуть в (НРЗ 48:02) args путем многоточия.

Здесь у нас вызывается метод ScanAll и что самое интересное: после процессинга этот Rows, я напоминаю, что я говорил, что в базе…, база не выдает все значения сразу, она может делать какой-то свой внутренний пейджинг, чтобы не гонять огромный данные по сети. И поэтому Rows у нас является некоторым таким интерфейсом, который надо закрыть после того, как вы получили оттуда все данные. И в…, говоря про базы данных, я этого не показывал и это достаточно важно. Если вы работаете с сырым соединением с базой данных, которая из расчёта такого подобного рода Rows закрываемые – вам нужно обязательно их закрыть после процессинга, иначе у вас в соединении останется открытым этот пагинатор и уже до истечения таймаута соединения с базой, этот пагинатор у вас останется открытым, и это соединение использовать уже не получится. Поэтому его обязательно надо закрыть, что и библиотечка pgxscan делает за нас. В случае любых запросов с многими результатами – обязательно их закрывайте, если вы не используете никакие writer-ы.

Здесь мы, соответственно, передаем результаты. Напоминаю, что pgxscan (НРЗ 49:21) сам результаты в пользователя. Здесь у нас в структуре определены точно такие же поля. Если бы они не были бы определены – нам нужно было бы здесь написать db:«userid», что-нибудь в таком духе, то есть как-то бы определить, как это поле называется. Но, а здесь всё в порядке, и мы можем просто собрать пользователей и засунуть их в результат. Опять же, если ошибка не ноль, мы её возвращаем…, мы её выводим, но не возвращаем, потому что нам не очень

интересно, на самом-то деле, какая там ошибка. И по спецификации, по контракту, если какая-то ошибка произошла в момент выборки баз данных, мы просто показываем пустой список. Я так решил. В реальной жизни, естественно, вы можете эту ошибку вернуть и вывести.

Соответственно, получение пользователя по id-шнику, то же самое: здесь мы получаем пользователя, здесь мы можем сразу собрать забросы, никак у нас не будет меняться. Передать туда просто параметр id и на этом закончить.

Создание пользователя: здесь мы уже работаем непосредственно с Pool-ом, делаем Exec и создаем пользователя с 2-мя параметрами – именем и рангом из модели, опять же, пользователя, которая к нам пришла извне. В принципе, с пользовательским storage-м это всё.

Машинный storage у нас немножко более интересен, потому что помимо того, что мы в списке получаем список всех машин, мы ещё и получаем с-join-енный список, то есть я про это тоже уже говорил: мы соединяем 2 таблицы для того, чтобы получить данные о пользователях, которые привязаны к конкретной машине для того, чтобы у нас наш список мог выглядеть вот так. То есть у нас есть машина, эти все параметры, все данные у нас приходят из непосредственно таблицы автомобилей, colour, brand, license_plate, id. Но по Userid мы забираем owner-а. То есть мы не просто пользователю показываем какой-то user id, а мы ещё и показываем, что этого owner-а зовут Name и у него есть какой-то…, что этого пользователя зовут «Lark», и у него есть какой-то ранг. То есть мы забираем эти данные из связанной таблицы.

И я напоминаю, что postgres устроен таким образом, что он возвращает плоскую структуру всегда, а у нас здесь структура, как можно видеть, вложенная, то есть здесь 1 объект и в него вложен другой объект. Таким образом, когда…, то есть

здесь опять же у нас всякие фильтры, здесь я уже использую ILike кроме users.id, id – у нас строгое соответствие. Соответственно, здесь я использую промежуточный так называемый буферный тип dbResult, в смысле userCar. UserCar – это соединенные параметры. Здесь можно как раз увидеть mapping, потому что UserId и User с большой буквы Id pgxscan не распознает. То же самое с «carid», то есть здесь у нас эти соединенные параметры – они отдельно map-ятся.

И эта структура будет в дальнейшем использоваться для того, чтобы из неё, из плоской структуры собрать нашу вложенную структуру, то есть наша model car имеет Id, Colour, Brand, LicencePlate – обычные параметры, и имеет owner-a, который является моделькой пользователя. И отсюда мы можем собрать модельки машины, модельку пользователя исходя из нашей плоской структуры, то есть это такой обычный, по сути, говоря, конвертер. Таким образом, здесь мы собираем результаты в промежуточном варианте, прогоняем их через наш конвертер и возвращаем.

Это не часть логики, опять же, процессора, хотя казалось бы – это достаточно сложный такой логический кусок, потому что processor может работать с любой базой данных. Вы можете хранить, например, данные в (НРЗ 53:31), где они уже будут выложены и тогда вам ничего конвертировать не надо. Значит, эта ответственность – эта ответственность непосредственно самого storage-a.

Таким образом такие куски логики должны включаться в сам storage, то есть валидация, что запросы консистентные. Сборка параметров из плоской структуры выложена так, как удобно вам с ней работать по контракту – должна выполняться непосредственно storage-ом самим. То есть моделькой, то есть структурой и методами для работы именно с базой данных.

Последнее, о чём хотелось бы сказать: здесь можно видеть, что у меня здесь определены json-ы. Json-ы нужны для (НРЗ 54:18) для того, чтобы правильно именовать поля. То есть если я уберу отсюда json-ы, то эти поля – они будут начинаться с большой буквы: Id с большой буквы, Name с большой буквы, Range с большой буквы. В принципе, ничего страшного, но это будет непривычно особенно на фоне того, что в целом в json-е принято именовать такие поля с маленькой буквы – это такое правило хорошего тона. Поэтому здесь я просто в модельках, которые потом Marshal-ятся, определяю уже сам непосредственно json для mapping-а полей, для имен этих самых полей.

В принципе, мы посмотрели почти всё. То есть все наши слои, модельки, handler-ы, processor-ы, как можно видеть довольно небольшое приложение. Обычно с ростом сложности приложения, увеличиваются все слои, но хотелось бы обратить внимание на то, что, как правило, handler-ы и storage-ы, то есть базы данных – они не содержат в себе никаких сильных вложенностей. И если у вас, например, что-то надо скомбинировать из 2-х storage-й, вы скорее, создадите отдельный processor и будете его использовать в другом processor-е. То есть processor-ы могут создавать довольно сложные связи. Как правило, handler-ы и базы данных не создают таких сложных связей. Здесь у меня из сложных связей есть только handler-ы, общие методы для всех handler-ов, где я оборачиваю ошибки и результаты OK.

Последнее, о чём я не рассказал – это мой middleware. Он подключается здесь в route-х, его можно было бы подключить в непосредственно создание этих route-ов, но я предпочитаю мой middleware-ом видеть отдельно от роутеров. Ни то, ни другое не является такой критической серьезной ошибкой. Соответственно, Middleware у меня просто log-ирует запросы, здесь можно увидеть info msg, что там выполнено, какой запрос и какой у него метод. А дальше я обязан в middleware-е, то есть у него свой интерфейс, в него передаются handler-ы – это

такая цепочка, то есть первый…, то есть сервер передаёт 1-му handler-у в очереди. 1-й handler в очереди что-то сделал, передаёт 2-му и так далее. Если вы этого не будете делать, ваш handler остановится и ничего не напишет в ResponseWriter, например, как этот handler, то есть он ничего не пишет в ResponseWriter. У вас ничего не будет выводиться, поэтому и handler просто печатает в log и прокидывает дальше.

Этим удобно пользоваться, например, для авторизации. Вы можете написать handler авторизации, который будет лазить в базу данных и, например, проверять, если в handler-ах есть какой-то Authkey, какое-то значение. И он может сходить в базу, посмотреть, есть ли такой Authkey или сходить в какой-нибудь Redis и так далее. И если его нет, то сразу здесь вернуть ошибку, а если есть – то продолжить дальше. В принципе, примерно для этого handler-ы и используются.

И давайте посмотрим, как работает сервис, соответственно, список с машинами вы уже видели. Опять же, invalid header name – нет такого header-а. Опять же, эту ошибку тоже было бы неплохо обработать. Вот наш список с 200, то есть здесь у нас id-шники наших машин, здесь у нас машины как раз и собрана из такой-то (НРЗ 58:03) – всё собрано из плоских структур в выдающиеся структуры – в структуры с нашими owner-ами.

Опять же, Get car by id – «not found», потому что я передаю метод Post. Опять же, давайте посмотрим «not found», да, вот 404 – «not found». Соответственно, передаем метод get, нахожу человека. Здесь, в принципе, никак ничем не отличается…, нахожу машину, вернее, здесь никак ничем не отличается, в принципе, от списка, кроме того, что у нас здесь объект. Опять же, зачастую в api-шках по id-шнику выдается больше информации, чем в списке. Например, если бы у меня была б машина и ещё какой-то статистика по посещениям, можно было бы выдать в id-шнике. В списке это было бы слишком накладно, потому что для

каждой машины надо было бы запрашивать этот список, его разворачивать и так далее.

Опять же, Create car, метод для сборки автомобиля, сюда передаётся объект. Мы можем сюда что-нибудь новенькое передать. Опять же, это отдельно можно проверить, что у нас номера соответствуют каким-то кадрам – это всё будет делаться processor-ом. И получаем результат «OK» либо результат негативный, если у нас чего-то не хватает, например, у нас нет номера. Это всё выдаётся processor-ом, прокидывается дальше обратно в handler и этим самым handler-ом оборачиваются в ошибку bad request. То же самое со списком пользователей, с его фильтрацией. с нахождением пользователя по id. Вот у нас пользователь не найден, такого пользователя нет, пользователь 1 точно есть. Создание пользователя, опять же.

И последнее, что хотелось бы показать – это сочетание фильтров. То есть я, например, хочу найти все машины с brand-ом, начинающимся или вообще с brand-ом, содержащим в себе слово «B» и цветом, например, green. Таким образом я нахожу Subaru. И 2 Subaru-ы, значит, зеленого цвета у двух разных людей. Если я передам сюда какую-то дичь, то не получу ничего, и это абсолютно нормально.