Мы посмотрели, как тестировать код, который непосредственно зависит от нашей базы данных. То есть, по сути, мы здесь тестировали наши запросы в базу данных. Но часто бывает так, что на слой уровнем выше мы используем методы наших репозиториев. И нам, на самом деле, не важна реализация.

То есть, допустим, у нас есть метод GetItemTitleByID. Этот метод не является методом репозитория, а это какой-то сервисный метод, который, например, используется в вызовах API, или в каких-то handler-ах наших. Внутри этот метод, конечно же, использует репозиторий, вызывает метод, который уже взаимодействует с базой данных. Но есть кейсы, когда нам не важно, что там внутри этого метода, мы не хотим знать реализацию, что там внутри, а сказать просто, что мы хотим проверить, что метод GetItemTitleByID сработает корректно, если, например, результат вот этого метода будет такой, какой мы предоставим. Но при этом нам не важно, что происходит внутри этого метода.

Для того, чтобы это проверить, нам необходимо построить архитектуру на наш код таким образом, чтобы у нас была не такая большая связанность, и по возможности вынести всё под интерфейсы. Поэтому в данном случае мы реализовали структуру нашего сервиса ItemService, и в этой структуре у нас есть поле ItemRepo. То есть, смотрите, у нас есть структура сервиса, которая зависит от какого-то репозитория. И этот репозиторий, на самом деле, является интерфейсом, который вот здесь же и объявлен. У нас есть ItemRepoInterface, который говорит о том, что необходимо иметь под своей реализацией функцию GetByID, которую, как мы видели, и содержит наш репозиторий. И вот IDE мне говорит, что есть реализация в нашем репозитории. Но на самом деле, смотрите, это совсем другой пакет. Функция GetByID объявлена в пакете repo. А наш сервис лежит в пакете service. И они друг о друге ничего не знают. Поэтому для создания нашего сервиса, нам нужен метод NewItemService, который параметром как

раз-таки получает repo соответствующий интерфейсу, объявленному тут же. И в результате мы, конечно же, возвращаем структуру нашего сервиса.

Внутри этого метода конструктора ничего нового нет. Здесь просто создается наша структурка, и мы внедряем в неё нашу зависимость, а именно репозиторий, который снаружи передаём внутрь. И, как вы уже видели, метод GetItemTitleByID – это метод нашей структуры ItemService, внутри он просто вызывает метод GetByID, который есть в интерфейсе, получает ответ, проверяет на ошибку и, если никаких ошибок нет, то возвращаем Title.

Смотрите. Для того, чтобы тестировать базу данных, мы использовали пакет sqlmock. Но в этом случае sqlmock нам ничем не поможет. Здесь мы должны сделать заглушку не для базы данных, а для интерфейса. По сути, нам нужно создать заглушку для вот этого вот интерфейса. Потому что нам не важна будет реализация, нам просто нужно сказать, что метод вернёт то или иное значение.

Для того, чтобы создавать такие mock-и, нам необходимо воспользоваться пакетом go mock. Вы видите страничку репозитория этого пакета, и как вы могли заметить, он находится в официальном репозитории языка go. Здесь также достаточно подробная документация есть. И для того, чтобы установить go mock, нам нужно вызвать go install, go mock, mockgen и выбрать определенную версию. У меня, на самом деле, на компьютере уже go mock установлен. Давайте посмотрим, что можно с ним сделать.

Здесь у меня уже сгенерирован mock-файл, давайте я его удалю. Смотрите, чем помогает нам go mock? Здесь я написал Makefile, в котором поместил команду mock. И mockgen, как раз, это то, что нам необходимо скачать со странички репозитория. Вызывается команда mockgen, где я указываю source, то есть я говорю команде, что мне нужен файл по вот такому пути. Результат, который

получится в результате команды mockgen, необходимо положить в файлик item_mock.go. Всё это будет происходить в пакете service, и название интерфейса, которого мне нужно за-mock-ать.

И смотрите, я вызываю команду mock. И здесь у меня появляется item_mock. И на самом деле, этот файл представляем собой mock нашего интерфейса, который описан вот здесь. Теперь, что мы можем сделать? Мы можем начать писать наш тест, абсолютно также, как мы писали до этого. Мы пишем тест-кейс, начинаем с названия Test, и даём название GetItemTitleByID. А дальше создаем контроллер для gomock-а. Это похожий контроллер, как мы создавали для базы данных, который поможет нам осуществлять некоторые проверки нашего mock-а. Создали наш контроллер. И в defer обязательно нужно его завершить вызовом метода Finish.

Смотрите, дальше мы создаем наш репозиторий. А создаём мы его с помощью метода, который появился в генерированном файле. То есть этот сгенерированный метод создаст нам mock нашего интерфейса. После того, как мы вызвали этот метод, передали в него наш котроллер, в переменной repo у нас будет лежать структура, которая соответствует нашему интерфейсу ItemRepoInterface. И тогда мы с вами можем уже создать нашу структуру service. Для этого нам достаточно просто вызвать метод, который уже мы объявляли с вами самостоятельно, и передать туда repo, который является mock-ом. Но так как он соответствует интерфейсу, всё замечательно проходит.

Далее, мы по аналогии, как делали это для базы данных, описываем, что мы ожидаем в результате. Ну, здесь мы просто взяли Item, который также будет с ID:10. И с Title-ом: TestTitle. И говорим нашему репозиторию, что мы от него ожидаем. А ожидаем мы вызов метода GetByID, в который передадим параметр 10. И ожидаем, что этот метод вернёт нам структуру Item с ID:10 и Title: “TestTitle”.

Ну, и в качестве ошибки вернется nil. Так как у GetID, как вы помните, два возвращаемых значения.

Сейчас я перейду еще раз к реализации. То есть, GetByID возвращает нам структуру Item и ошибку. В этом случае у нас ошибки не будет, мы вернём nil, а структура Title вернется та, которую мы здесь описали, то, что мы и ожидаем.

И дальше совершенно спокойно мы можем вызвать метод нашего service-а, то есть структуру service-а, которую создали вон здесь - GetItemTitleByID. Это уже метод, который написанный нами, внутри он вызовет интерфейсный метод и всё пройдет гладко. Передаём туда параметр, который и ожидаем к передаче. Дальше просто проверяем на предмет ошибок. Если есть ошибка, то выходим. Если ошибки нет, значит, мы получили какой-то результат. Нам осталось сравнить значение этого результата, который вернул наш метод service-а, со значением результата Title, который мы ожидаем увидеть в итоге. И если всё хорошо, то тест пройдет успешно.

Давайте запустим теперь этот тест. Делается это так же: с помощью команды go test, но нам нужно указать другое расположение наших файлов. Данные файлы у нас лежат в папке service. Как мы видим, всё тоже прошло успешно. В формате verbose мы видим название методов, которые запускались. Если, допустим, мы здесь передадим не 10, а что-нибудь другое, то, конечно же, получим ошибку. И здесь мы говорим, что вызов метода не соответствует тому, что мы ожидаем. Потому что, смотрите, если мы передадим в наш сервисный метод 5-ку, то внутри метода наш метод GetByID вызовется со значением 5. Но мы до этого в нашем mock-е указали, что мы ожидаем, что GetByID вызовется со значением 10. И тогда мы не оправдаем наших ожиданий. И наш тест провалится.

Давайте вернём. И теперь мы можем запустить наши тесты все вместе. Для этого нужно сделать следующее, чтобы тест запускался во всех директориях. И как мы можем видеть, у нас сфейлился один тест repo, который мы намеренно изменяли для того, чтобы посмотреть ошибку. Здесь, конечно же, нужно передать 10. И в этом случае все тесты пройдут успешно.