Итак, мы с вами написали тесты для простой функции fib. Эта функция не связаны ни с каким другим кодом. И у нее нет никаких зависимостей. Поэтому написать тест для такой функции не составляет особого труда. Но на практике в наших проектах код, на самом деле, сложнее и состоит из разных слоев. Давайте попробуем написать тесты для нескольких таких слоев.

Рассмотрим слой репозиториев. Я создал папку repo, где привел пример работы с базами данных. Допустим, у нас есть репозиторий, который мы создали, то есть что такое репозиторий? Это некоторая структура, из которой мы входим в нашу базу данных и получаем данные. Видим, что у нас есть структура Repository, в ней есть поле db, где мы держим структуру, отвечающую за соединение с нашей базой данных. По традиции, у нас есть метод конструктор NewRepo, куда мы передаем наш connect, и создается структура репозитория.

У данной структуры есть метод GetByID. Ну, давайте представим, что нам что-то требуется достать из базы данных по идентификатору. Мы пишем метод GetByID и передаем в параметры идентификатор. И в результате функция нам вернет структуру Item, давайте посмотрим на нее, я ее реализовал в пакете entity. Item у нас состоит из двух полей это идентификатор и Title. И вот здесь вы видите, что в базе данных это связано с одноименными полями в том числе. Вернемся в репозиторий. И вторым параметром у нас репозиторий возвращает, то есть метод GetByID возвращает ошибку.

Давайте смотреть на реализацию. Здесь мы производим запрос в базу данных с placeholder-ом и с условием where. И дальше передаем туда идентификатор, который заменится на вот этот placeholder. Получается, что в базу данных полетит условие «выбери мне идентификатор и Title из таблицы items», где идентификатор равняется переданному значению в наш метод GetByID. Здесь мы сразу создадим пустой результат Item-а. И дальше проводим проверки. Соответственно, если у нас

вышла ошибка, то мы выдадим пустой Item и саму ошибку. Если ошибок никаких нет, нам обязательно надо закрыть наш rows. И после этого, мы в цикле проходим по результату, который вернулся из базы данных. И начинаем сканить наш результат в структуру.

То есть что делает метод Scan? Метод Scan сканирует результат, который пришел из базы данных, и в наше поле структуры, соответственно, парсит ответ. В данном случае, мы заполним из базы данных, так как у нас запрашивается ID и Title, соответственно, будут заполнены поля ID и Title нашей структуры. И, если здесь тоже происходит какая-то ошибка, мы вернем пустой item и ошибку. В результате, при успешном прохождении всего цикла, мы вернем заполненный item результатами из базы данных. И ошибка в данном случае будет – nil.

Итак, предположим, у нас есть вот такой репозиторий, и мы хотим написать тесты. Как нам быть, ведь в данном методе мы не можем просто написать тест, как в случае с методом fib. Потому что в методе fib, внутри функции, мы ни на что не завязаны. Она вызывает саму себя и нет никаких зависимостей. Но вот в методе GetByID нам нужно каким-то образом прикрутить базу данных, потому что мы осуществляем запрос к базе.

На самом деле, можно поступить следующим образом. Смотрите. Вот эта структура является структурой абстракции над соединением баз данных. Там, на самом деле, может находиться и база MySQL, и база PostgreSQL, и другие SQL-подобные базы данных. Поэтому в наших кейсах было бы неправильно завязываться на какой-то конкретной реализации. Для того, чтобы с этой ситуацией справиться, нам нужны, так называемые, заглушки (ну или Mock-и).

Для базы данных есть такая библиотека, которая называется sqlmock. Давайте пройдем на сайт этой библиотеки, в репозиторий на GitHub.com. И увидим, что

здесь достаточно много описания, есть документация и для того, чтобы нам установить ее, надо вызвать go get и наименование репозитория. Давайте произведем это еще раз, для наглядности. После этого данный пакет скачается и будет доступен для импорта, что здесь и происходит.

Что нам дает этот пакет go-sqlmock? Смотрите, здесь у меня написан тест-кейс TestGetByIDSuccess, что означает, что мы, подобно как писали это для функции fibonacci. Видите, у нас здесь было три выхода из функции, и мы для всех трех веток писали отдельные тест-кейсы, то есть кейс первый, второй и третий. В этом случае, с репозиторием, похожая ситуация. Смотрите. У нас есть первый выход, когда мы сразу получаем ошибку. У нас есть второй выход и у нас есть третий кейс, когда все прошло успешно. В данном случае, у нас тест-кейс написан, как раз, на третий вариант, когда все проходит успешно. Абсолютно также определяется функция, передается в нее структура testing.T из пакета, уже знакомого, testing. И дальше уже происходит работа с sqlmock-ом.

Смотрите. Мы вызываем функцию sqlmock.New. В данном случае, мы создадим новый connect, но это уже будет не реальная база данных, а, так сказать, виртуальная. То есть все запросы, которые полетят в этот открытый connect, ну назовем это connect, так как это происходит виртуально, на самом деле, это не совсем соединение. Мы получаем эту структуру первым параметром, первым результатом вызова функции new. Во втором результате приходит mock, это некий контроллер, который позволяет нам управлять и осуществлять проверку данных. Я чуть позже об этом скажу. И дальше стандартный результат ошибки. Если у нас произошла ошибка, то мы сразу заканчиваем наш тест-кейс. И говорим, что не получилось создать mock базу данных. После этого нам обязательно нужно закрыть нашу базу данных. Поэтому в defer мы вызываем метод Close.

Итак, смотрите, у нас вот этот объект db, структурка, она соответствует тому, что нам нужно для создания нового репозитория. Поэтому мы можем создать заглушку для sql.DB в нашем тесте. И в функцию NewRepo передать как раз уже его. То есть таким образом, мы подменяем реальную реализацию MySQL соединения на sqlmock, который внутри себя производит похожие операции.

Что мы делаем дальше? Дальше мы объявляем наши колонки, которые будут виртуально храниться в нашей виртуальной базе данных. Мы говорим, что нам нужны NewRows. И объявляем, что наша колонка ID и колонка Title. Ну, так как нам больше ничего не нужно, остановимся на этих двух вариантах. Потом мы объявляем структуру, которую мы ожидаем увидеть в результате. Допустим, давайте определим структуру item с идентификатором 10 и Title-ом TestTitle. Что происходит дальше? А мы просто добавляем в нашу виртуальную табличку идентификатор 10 и наш Title TestTitle. То есть вот здесь мы объявили колонки, а вот здесь эти колонки заполнили данными.

Дальше, как раз, мы работаем с mock-ом. Mock, который мы получили в результате вызова New, имеет некоторые методы для работы с виртуальной базой данных. Здесь мы вызываем функцию ExpectQuery. Она означает следующее. То, что мы передадим в параметрах этой функции будет ожидаться в запросе к базе данных. То есть, смотрите, при вызове функции GetByID, в нашу базу данных полетит вот такой запрос. И, как раз-таки, его нам и нужно протестировать. И функция ExpectQuery, как раз-таки, принимает параметр и говорит, какой вы запрос ожидаете в результате отправки в базу данных. И функция ExpectQuery, как раз-таки, говорит нам – напиши запрос, который мы ожидаем увидеть на выполнение в базе данных. То есть здесь мы передаем запрос, который мы хотим в результате получить в базе данных, и передаем параметр, который должен вызваться в запросе. То есть как у нас в методе Query, у нас есть запрос и дальше параметр.

Таким образом, настроив ожидаемый запрос и ожидаемый параметр, мы в итоге говорим, что метод должен вернуть в результате вот такие поля, которые мы до этого создали в нашей виртуальной базе данных. Дальше мы, уже с объявленным нашим репозиторием, который просто создает нашу структуру, это уже код, написанный нами, вызываем просто метод GetByID. И передаем ему идентификатор 10. Помним, что по этому идентификатору мы добавили в нашу виртуальную табличку строку, потом проверяем на ошибки. Если ошибок нет, следующим шагом необходимо проверить, оправдались ли наши ожидания, которые мы настроили вот здесь вот. То есть на этом шаге произойдет проверка нашего запроса, совпадает ли переданный в запрос параметр идентификатора с тем, что получили при вызове нашего метода GetByID. И если все хорошо, значит мы в результат сюда получили нашу структуру item.

И нам осталось просто сравнить результат выполнения нашего метода GetByID с тем, что мы условились ожидать от этого метода. То есть у нас item.ID должен равняться expect.ID. И item.Title должен равняться expect.Title. Если же они не равняются, то мы выводим ошибку.

Давайте запустим этот тест и посмотрим на результат. Так как этот тест находится в другой папке, давайте укажем, где конкретно нам нужно искать тестовые файлы. Как мы видим, все тесты прошли. И тест-кейс считается успешным.

А если бы мы, например, здесь передали другой идентификатор, но в нашей виртуальной базе данных у нас идентификатор равняется 10, то мы получили бы ошибку. Потому что мы не нашли бы в этом случае, нужные нам данные, и они бы здесь не сматчились. Но и даже до того, как мы дошли до этой проверки, проверка зафейлилась вот на этом этапе. Потому что мы говорим, что аргумент не соответствует тому, что мы ожидаем. Так как мы здесь говорим, что метод вызовется с аргументом 10, а на самом деле мы в GetByID передаем 7. И тогда

уже внутри метод Query вызовется с аргументом 7. И данная проверка уже не пройдет.

Таким образом, мы можем написать тест-кейсы для остальных случаев, когда метод нашего репозитория будет возвращать ошибку. Для ошибки здесь есть метод WillReturnError, и мы можем описать, какой нам нужна ошибка. И таким образом проверить все точки выхода и все ветки нашего репозитория, метода нашего репозитория.