Итак, какие у нас есть стандарты организации кода? Уже говорил, подчеркну еще раз, это очень важно. Один файл исходника содержит решение конкретной задачи. Мы там валидируем какие-то поля в них или одно какое-то поле, мы какой-то config там собираем и оборачиваем какой-то ответ, и так далее. Не решаем несколько задач. Один или несколько файлов исходников, которые как-то объединяются по общим группам, например, все валидаторы, вся работа с файлами и так далее. Объединяются в Package. То есть Package, например, files, будет у нас отвечать за работу с файлами. Package validation будет отвечать за работу со всей валидацией. И, значит, комплект из Package-ей, которые решают какую-то общую задачу, которую мы назвали бы библиотекой или приложением — это уже модуль. Это объединение, которое можно использовать от и до для решения какого-то класса проблем, там все про работу с PostgreSQL, все про работу с Kubernetes API, все про, значит, авторизацию в нашей системе, ну и так далее. То есть наши сервисы, которые мы пишем, наши приложения, которые мы пишем — это все будут модули.

И модули мы можем также скачивать и использовать в своих проектах, как библиотеки. То есть это, в принципе, в терминологии go одно и тоже. Если у нас есть какое-то там приложение готовое или у нас есть какая-то библиотека, которая нам просто помогает работать, и то, и другое будет модулем. И что самое важно, модули могут и, скорее всего, будут зависимы друг от друга, но это должно быть явно указано. Если вы используете какие-то библиотеки, вы пользуетесь, значит, строчками requirements, и, значит, скачиваете чужие модули. И если вы заливаете свои работы на Github, авторы других, значит, авторы, использующие вашу библиотеку, или авторы, использующие ваше приложение, другие программисты, они не думают о том, какие модули они могут быть использованы.

А вот как ими пользоваться, как раз сейчас мы и посмотрим. У меня есть какое-то мое приложение, соответственно, мое приложение состоит из 1 файлика main go в

Package main, импортирующая кучу файликов. Вот можно видеть, что оно импортирует, соответственно, всякие различные модули k8s.io/client-go, k8s.io/apimachinery, что (НРЗ 02:20-02-23) go-чным клиентам. И, соответственно, можно видеть, что я здесь выполняю какие-то действия с уже, непосредственно, самим клиентом. То есть я здесь вгружаю конфигурацию из моего (НРЗ 02:34), дальше я вывожу список всех pod-ов, об этом мы поговорим немножко позже в курсе лекций. В общем, у меня есть какое-то приложение. Если я сейчас прям сделаю go build, то, естественно, у нас будет куча ошибок, потому что нет таких Package-ей, нет их в Package, то есть в той папочке go, нет их нигде сохраненных. Значит, соответственно, каким-то образом, нам их нужно получить.

Как мы это сейчас можем сделать? У нас есть великая и ужасная команда go mod, соответственно, go help mod, простите. Соответственно, мы вводим команду, и она нам, что позволяет делать? Она нам позволяет скачать модули, подредактировать go.mod каким-то образом, вывести graph всех этих модулей.

Но если вы работали с (НРЗ 03:27), то скорее всего, вам будет это уже знакомо. Либо с pip-ом Питоновским, либо с (НРЗ 03:32), ну там короче все, в принципе, одно и тоже. Но пока у вас есть просто файлик main.go, это файлик main.go. Нам нужно из него сделать модуль. Нам нужно, соответственно, как-то…, какой-то создать файл, в котором мы опишем все зависимости, и в котором мы опишем, как работать нашему модулю. Соответственно, мы делаем go mod init. Он у нас падает, потому что он нам говорит, что, значит, нам нужно специализировать путь к модулю, то есть нам нужно понять, как этот модуль будет называться для go, то есть указать его, так сказать, путь. Если у нас есть репозиторий на GitHub-е, он возьмет, соответственно, путь с GitHub-а. То есть, если бы у меня здесь был инициализирован git, он бы взял модуль git-а. Но git у меня не инициализирован, поэтому, значит, мы здесь пишем просто init, не знаю, lectures ories example.

Итак, он нам, соответственно, создал модуль, он его назвал, как мы можем видеть. То есть такие же вещи мы можем видеть в GitHub-е во всех модулях. Вот они. Это, соответственно, путь к этому модулю. Так, как мы будем к нему, непосредственно, обращаться. И соответственно, создал версию go, от которой этот модуль компилируется, то есть, если здесь вы собираете проект с модулем версии go, ниже, чем указанная, он ругнется на то, что, значит, компилятор ниже. Но это достаточно очевидно.

Но, как можно видеть, просто создание файлика go mod, нам сильно не поможет, потому что у нас нет все равно этих модулей. Но теперь ошибка другая. То есть он говорит, что никакие модули, которые мы требуем, которые у нас лежат в зависимостях, не предоставляют нам вот такой вот Package k8s.io/apimachinery/pkg/api/errors. Можно, кстати заметить, что вот это — это путь модуля, а вот это — это уже его подпапочка, соответственно, на этот Package. Соответственно, да, действительно, нет у нас никаких requirements, потому что мы их не записали. Можно руками, конечно, это все записать, можно сделать там go get, сохранять эти все модули, он их будет качать в Package, и как-нибудь у вас приложение соберется.

Но есть путь лучше, есть путь, соответственно, go mod 1. Он сам вам найдет эти модули, которые у вас указаны в импортах, потому что эти импорты — они, собственно говоря, абсолютные. То есть видно, что у вас все модули указаны как бы абсолютным путем. То есть это физический путь к папке какой-то. Вот он, соответственно, go их все нашел, положил их в go sum. Go sum – это, по сути говоря, такие как бы хэши ваших файлов, для того, чтобы, если вдруг у вас что-то посыпалось или репозитории какие-то недействительные, вы бы об этом узнали. То есть, если, например, у вас появился goproxy в системе глобальной, и у вас все форвардится на этот goproxy, вы бы хотя бы знали, что у вас по какой-то причине не может что-то собраться.

Этот go sum делается самим go mod-ом, его менять не надо. Если вы заливаете проект, лучше залейте этот go sum. Это позволит другим разработчикам точно скачать ваши зависимости и их, соответственно, использовать, точно зная, что это те же самые зависимости, которые у вас на компьютере.

Соответственно, теперь у нас все готово к запуску. У нас есть go mod, go sum, то есть, есть руководство go, откуда и как брать эти самые файлики, есть все зависимости. Здесь, кстати, будут указаны еще все зависимости ваших библиотек со всеми версиями. Вот можно здесь увидеть, да? И это важно, потому что у вас в разных модулях могут использоваться разные версии библиотек, ну то есть разные версии других модулей, если пользоваться терминологией go. Соответственно, мы готовы, мы можем запускать. Go build main go, скачать модули автоматически, если у вас есть go sum. То есть вам не надо никакие дополнительные команды запускать. И все. У меня все запустилось, конечно, упало, потому что файлика нет у меня такого с конфигурацией Kubernetes-a. Но ничего страшного. Важно, что это все собралось, и теперь любой разработчик, который мою программу скачает, он сможет…, вернее, (НРЗ 07:53) с GitHub-a, он сможет ее просто взять и собрать с помощью Go build main go или go run main go. Соответственно, в принципе, это все. То есть вот мы в 3 простых команды подтянули все зависимости, запустили все. И соответственно, довольные собой, собрали проект.

Последнее про что я хотел бы поговорить — это так называемый, vendoring.