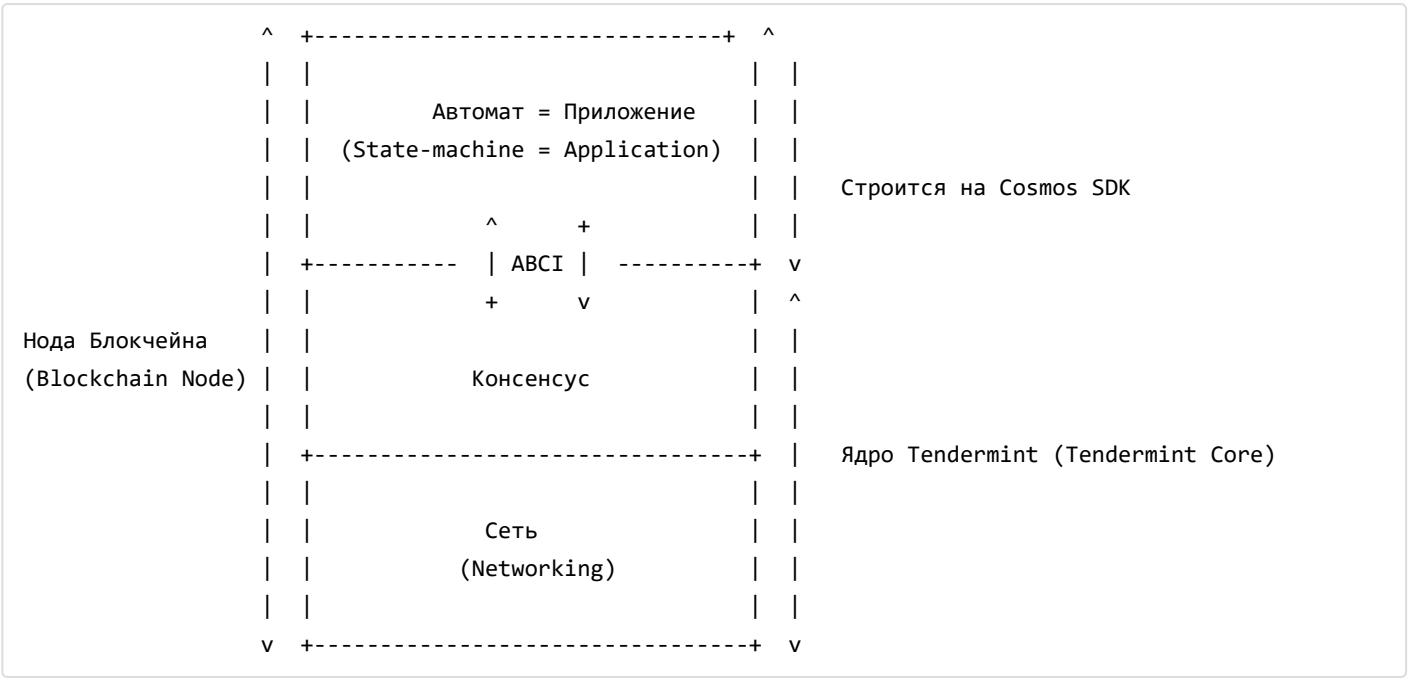
Анатомия SDK Приложения

Клиент Ноды

Демон, или Клиент Полной Ноды (Full-Node Client), - это центральный, основной процесс блокчейна на основе данного SDK. Участники сети запускают этот процесс для инициализации своей машины состояний (автомата - state-machine), соединенения с другими полными нодами и для обновления своей машины состояний при поступлении новых блоков.



Полная нода блокчейна выглядит как бинарник, двоичный файл, имя которого обычно заканчивается на `-d`, чтобы обозначить, что это демон (например, `appd` для `app` или `gaiad` для `gaia`). Этот бинарник собирается вызовом функции `main.go`, которая находится в `./cmd/appd/`. Это действие обычно делается с помощью Makefile.

Когда главный бинарник собран, ноду можно запускать командой `start`. Эта команда делает три основные вещи:

1. Создаёт инстанс машины состояний, определённой в файле `app.go`.

2. Инициализирует эту машину состояний последним известным состоянием, извлекаемым из базы данных `db`, которая хранится в директории `~/.appd/data`. В этот момент автомат (state-machine) (машина состояний находится на высоте `appBlockHeight`.

3. Создаёт и запускает новый Tendermint инстанс. Помимо прочего, эта нода будет выполнять хендшейк со своими пирами. Она будет получать от них последнюю высоту блока `blockHeight` и проигрывать (replay) блоки для синхронизации с ней, если она больше локальной высоты `appBlockHeight`. Если `appBlockHeight` равна `0`, то нода стартует с генезиса, и Tendermint посылает через ABCI приложению `app` сообщение `InitChain`, которое запускает инициализатор `InitChainer` .

Основной Файл Приложения

В целом, основное определение машины состояний содержится в файле `app.go`. Главным образом, оно содержит **определение типа приложения (type definition)** и функции для его **создания и инициализации**.

Определение Типа Приложения

Первое, что определяется в `app.go`, - это `type` приложения. Обычно определение состоит из следующих частей:

- **Ссылка на `baseapp`.** Наше приложение, определённое в `app.go`, является расширение `baseapp`. Когда транзакция транслируется (relay, передаётся, ретранслируется) Tendermint-ом нашему приложению, наше `app` использует методы `baseapp`, чтобы направить (маршрутизировать, route) их соответствующему модулю. `baseapp` реализует бОльшую часть основной логики нашего приложения, в том числе все ABCI методы и логику маршрутизации.
- **Список ключей хранилищ (стор-ключей, store keys)**. Хранилище (store), где хранится полное состояние, реализовано в Cosmos SDK как мультистор (`multistore`) (т.е. хранилище хранилищ). Каждые модуль для сохранения своей части состояния использует одно или много хранилищ в этом мультисторе. Доступ к этим хранилищам осуществляется по специальным ключам, которые объявляются в типе приложения `app`. Эти ключи, как и `keepers` ("кошельки", или "хранители"), составляют основу, самоё сердце объектно-уполномоченной модели (object-capabilities model) Cosmos SDK.
- **Список `keeper`-ов модуля.** Каждый модуль определяет абстракцию под названием кошелёк, `keeper`, которая обрабатывает операции чтения и записи для хранилищ этого модуля. Методы `keeper`-а одного модуля могут вызываться другими модулями (если у них есть права). Именно поэтому они декларируются в типе приложения и экспортируются как интерфейсы в другие модули, чтобы эти модули могли обращаться к этим разрешённым (авторизованным) функциям.
- **Ссылка на кодек `codec`.** Кодек наше приложения используется для сериализации и де-сериализации структур данных при их хранении, поскольку в хранилищах могут сохраняться только значения вида `[]bytes`. Кодек должен быть детерминистическим. Кодек по умолчанию это amino.
- **Ссылка на менеджер модулей module manager** и на базовый менеджер модулей basic module manager. Менеджер модулей - это объект, который содержит список модулей нашего приложения. Это обеспечивает операции, относящиеся к этим модулям. Например, регистрацию маршрутов `routes`, query routes, или задание порядка исполнения между модулями для различных функций, таких как `InitChainer`,`BeginBlocker` and `EndBlocker`.

Пример определения типа приложения из `gaia` вы можете посмотреть в

+++
https://github.com/cosmos/gaia/blob/5bc422e6868d04747e50b467e8eeb31ae2fe98a3/app/app.go#L87-L115

Функция Конструктор

Эта функция создаёт (конструирует) новое приложение того типа, который был определён в предыдущем параграфе. Она должна соответствовать сигнатуре функции `AppCreator`, чтобы её можно было использовать в `start` command - команде старта демона нашего приложения.

+++ https://github.com/cosmos/cosmos-sdk/blob/7d7821b9af132b0f6131640195326aa02b6751db/server/constructors.go#L20

Вот главное, что делает данная функция:

- Создаёт инстанс (инстанцирует) новый `codec` и инициализируе кодек каждого модуля приложения с помощью basic manager
- Создаёт инстанс нового приложения со ссылкой на инстанс `baseapp`, кодек, и все нужные ключи хранилищ.
- Создаёт все кошельки `keeper`-ы, определённые в `type` приложения, с помощью функции `NewKeeper` каждого из модулей приложения. Учтите, что инстансы `keeper`-ов должны создаваться в правильном порядке, потому что `NewKeeper` может требовать ссылку на `keeper` другого модуля.
- Инстанцирует менеджер модулей приложения module manager с объектом `AppModule`каждого из модулей приложения.
- С помощью этого менеджера модулей инициализирует в приложении его маршрутов `routes` и query routes. Когда Tendermint транслирует (релеит) транзакцию данному приложению посредством ABCI, эта транзакция маршрутизируется к обработчику `handler` соответствующего модуля именно с помощью определённых тут маршрутов. Аналогично, когда приложение получает запрос (query), то этот запрос маршрутизируется к `querier` - обработчику запросов, - соответствующего модуля, также с помощью определённых здесь маршрутов.
- С помощью этого менеджера модулей регистрирует инварианты модулей приложения application's modules' invariants. Инварианты - это переменные (например, общий объём эмиссии токена), которые проверяются в конце каждого блока. Процесс проверки инвариантов выполняется посредством специального модуля, называемого `InvariantsRegistry` - Регистратором Инвариантов. Значение каждого инварианта должно быть равно его предполагаемому, заданному значению, определённому в модуле. Если это значение отличается от предполагаемого, запускается специальная логика, заданная в регистраторе инвариантов (обычно цепь останавливается). Полезно и важно обеспечить, чтобы никакая критическа ошибка не проскочила незамеченной и не вызвала далеко идущих последствий, которые будет трудно исправить.
- С помощью этого менеджера модулей устанавливает порядок выполнения между функциями `InitGenesis`, `BegingBlocker` и `EndBlocker` каждого из модулей приложения - application's modules. (Имейте в виду, что не все модули реализуют данные функции.)
- Задаёт остальные параметры приложения:
  - `InitChainer`: используется для инициализации приложения при его первом запуске.
  - `BeginBlocker`, `EndBlocker`: вызывается в начале и в конце каждого блока.
  - `anteHandler`: используется для обработки вознаграждений (fees) и верификации подписей.
- Монтирует (Mount) хранилища.
- Возвращает приложение.

Заметьте, что эта функция только создаёт инстанс приложения app, тогда как реальное (actual) состояние либо выводится из директории `~/.appd/data`, если данная нода рестартовала, либо генерируется из файла генезиса, если нода стартовала впервые.

Пример конструктора приложения из `gaia` можно посмотреть тут:

+++ https://github.com/cosmos/gaia/blob/f41a660cdd5bea173139965ade55bd25d1ee3429/app/app.go#L110-L222

Функция InitChainer

`InitChainer` - это функция, которая инициализирует состояние приложения из файла генезиса (т.е. балансов токена генезисных аккаунтов). Она вызывается, когда приложение получает от движка Tendermint сообщение `InitChain`, что происходит, когда нода стартует при `appBlockHeight == 0` (т.е. на генезисе).

Приложение должно задавать `InitChainer` в своём конструкторе constructor посредством метода `SetInitChainer`.

Как правило, `InitChainer` составляется главным образом из функции `InitGenesis` каждого модуля приложения.

Это делается вызовом функции `InitGenesis` менеджера модулей, которая в свою очередь вызовет функцию `InitGenesis` каждого из модулей, которые в нём находятся.

При этом порядок, в котором функции `InitGenesis` этих модулей должны вызываться, должен быть задан в менеджере модулей с помощью метода `SetOrderInitGenesis` этого менеджера модулей

Это делается в конструкторе приложения, и вызов `SetOrderInitGenesis` должен быть до вызова `SetInitChainer`.

Пример `InitChainer` из `gaia` тут:

+++
https://github.com/cosmos/gaia/blob/f41a660cdd5bea173139965ade55bd25d1ee3429/app/app.go#L235-L239

BeginBlocker и EndBlocker

Наш SDK даёт разработчикам возможность реализовать автоматическое выполнение кода как часть их приложения. Это реализуется через две функции, называемые `BeginBlocker` и `EndBlocker`. Они вызываются, когда приложение получает от движка Tendermint сообщения, соответственно, `BeginBlock` и `EndBlock`, что происходит в начале и в конце каждого блока. Приложение должно задать `BeginBlocker` и `EndBlocker` в своём конструкторе через методы `SetBeginBlocker` и `SetEndBlocker`.

Обычно функции `BeginBlocker` и `EndBlocker` главным образом и составляются из функций `BeginBlock` и `EndBlock` каждого из модулей приложения. Это делается вызовами функций `BeginBlock` и `EndBlock` менеджера модулей, которые в свою очередь вызывают функции `BeginBLock` и `EndBlock` каждого из модулей, которые он содержит. Надо иметь в виду, что порядок, в котором функции `BeginBLock` и `EndBlock` модуля должны вызываться, надо задать в менеджере модулей с помощью, соответственно, методов `SetOrderBeginBlock` и `SetOrderEndBlock`. Это делается посредством менеджера модулей в конструкторе приложения, и методы `SetOrderBeginBlock` и `SetOrderEndBlock` должны вызываться до функций `SetBeginBlocker` и `SetEndBlocker`.

Следует отдельно отметить, что важно помнить: создаваемые для приложений (application-specific) блокчейны детерминистичны. Разработчики должны внимательно следить, чтобы не вводить не-детерминистичные `BeginBlocker` или `EndBlocker`, а также чтобы не сделать их слишком дорогими в вычислительном смысле, потому что газ не ограничивает стоимость выполнения `BeginBlocker` и `EndBlocker`.

Вот здесь можно посмотреть пример функций `BeginBlocker` и `EndBlocker` из `gaia`:

+++
https://github.com/cosmos/gaia/blob/f41a660cdd5bea173139965ade55bd25d1ee3429/app/app.go#L224-L232

Кодек Регистратора

Функция `MakeCodec` - последняя важная функция файла `app.go`. Она предназначена для инстанцирования кодека `cdc` (например, amino), и инициализации кодека SDK и каждого модуля приложения с помощью функции `RegisterCodec`.

Для регистрации модулей приложения функция `MakeCodec` вызывает `RegisterCodec` на `ModuleBasics`. `ModuleBasics` - это базовый менеджер, - basic manager, - который составляет список всех модулей приложения. Он инстанцируется в функции `init()` и служит только для регистрации не-подчинённых (non-dependant) элементов модулей приложения (таких, как кодек). Подробнее о базовом менеджере модулей написано здесь

Вот пример `MakeCodec` из `gaia`:

+++
https://github.com/cosmos/gaia/blob/f41a660cdd5bea173139965ade55bd25d1ee3429/app/app.go#L64-L70

Модули

Модули - это сердце и душа SDK приложений. Их можно рассматривать как автоматы (state-machines) внутри нашего атомата (state-machine). Когда транзакция транслируется приложению из лежащего под всем этим Tendermint движка через ABCI, она маршрутизируется базовым `baseapp` к соответствующему модулю для её обработки. Эта парадигма позволяет разработчикам легко строить сложные автоматы (state-machines) за счёт того, что большинство нужных им модулей уже существуют. Для разработчиков основная часть работы по построению SDK приложения приходится на написание своих, новых кастомных модулей, требующихся для их приложения, поскольку этих модулей ещё нет, и интегрирование их с уже существующими модулями в одно согласованно работающее приложение. Стандартная практика хранить модули в папке `x/` в директории приложения (не путайте с папкой `x/` самого SDK, в которой находятся уже существующие, построенные модули).

Интерфейс Модуля Приложения

Модули должны реализовать интерфейсы interfaces, определённые в Cosmos SDK: `AppModuleBasic` и `AppModule`. Первый реализует основные не-подчинённые элементы данного модуля, например, `codec` , а второй обрабатывает основную массу методов модуля (включая те методы, которые требудт ссылок на `keeper`-ы других модулей). И тип `AppModule`, и тип `AppModuleBasic` определяются в файле `./module.go`.

`AppModule` даёт набор полезных методов для модуля, которые помогают сборке единого приложения из модулей. Эти методы вызываются из менеджера модулей - `module manager`, - который управляет всем набором модулей данного приложения.

Типы Сообщений

Сообщения, - `Message`s, - это объекты, определяемые каждым модулем, который реализует интерфейс `message`. Каждая транзакция `transaction` содержит одно или много `сообщений`.

Когда полная нода получает валидный блок транзакций, Tendermint транслирует каждую транзакцию приложению с помощью `DeliverTx`. После чего приложение обрабатывает транзакцию:

1. Сначала, по получении транзакции, приложение выводит (unmarshall) её из `[]bytes`.

2. Затем оно перед тем, как извлечь сообщение (или сообщения), содержащиеся в транзакции, проверяет несколько вещей относительно этой транзакции, как то выплату вознаграждений и подписи - fee payment and signatures, -

3. `baseapp` с помощью метода `Type()` в `message` маршрутизирует `message` обработчику `handler` соответствующего модуля, чтобы он обработал это сообщение.

4. Если сообщение обработано успешно, состояние обновляется.

Чтобы увидеть жизненный цикла транзакции более подробно, нажмите здесь.

Разработчики при разработке своих модулей создают свои типы сообщений. Принято начинать объявление типа сообщения с префикса `Msg`. Например, тип сообщения `MsgSend`позволяет пользователям передавать токены:

+++ https://github.com/cosmos/cosmos-sdk/blob/7d7821b9af132b0f6131640195326aa02b6751db/x/bank/internal/types/msgs.go#L10-L15

Оно обрабатывается `handler`-ом модуля `bank`, который в итоге вызывает `keeper` модуля `auth`, чтобы обновить состояние.

Handler, Обработчик

Обработчик, - `handler`, - это часть модуля, которая отвечает за обработку сообщения`message` после того, как его маршрутизировало `baseapp`. Функции `handler`-ов модулей выполняются только года транзакция транслируется из Tendermint ABCI сообщением `DeliverTx`-а. Если транзакция транслируется через `CheckTx`, то выполняются только stateless проверки и stateful проверки, связанные с вознаграждениями.

Чтобы лучше понять разницу между `DeliverTx` и `CheckTx`, а также разницу мжеду stateful и stateless проверками, нажмите здесь.

`handler` модуля обычно определяется в файле с именем `handler.go` и состоит из:

- **switch функции** `NewHandler` для маршрутизации сообщения к нужной `handler`функции. Эта функция возвращает `handler` функцию, и регистрируется в `AppModule`, чтобы её можно было использовать в менеджере модулей приложения для инициализации маршрутизатора приложения - application's router. Ниже приводится пример такой switch функции из пособия по службе имён - nameservice tutorial

  +++ https://github.com/cosmos/sdk-tutorials/blob/master/nameservice/x/nameservice/handler.go#L12-L26

- **Для каждого типа сообщений модуль определяет одну функцию - handler**. В этой функции разработчик пишет логику обработки данного сообщения. Обычно сюда входит выполнение stateful проверок для того, чтобы убедиться, что сообщение валидно, и вызов методов `keeper`-а для обновления состояния.

Функция обработчика возвращает результат типа `sdk.Result`, из которого приложение узнаёт, было ли сообщение успешно обработано:

+++ https://github.com/cosmos/cosmos-sdk/blob/7d7821b9af132b0f6131640195326aa02b6751db/types/result.go#L15-L40

Querier, Обработчик Запросов

Обработчики Запросов `Queriers` очень похожи на обработчики `handlers`, за исключением того, что они обслуживают пользовательские запросы к состоянию, а не обрабатывают транзакции. Запрос query инициируется из интерфеса конечным пользователем, который указывает маршрут запроса `queryRoute` и некие данные - `data`. Затем метод `handleQueryCustom` `baseapp`-а с помощью `queryRoute` маршрутизирует этот запрос к правильному `querier`-у приложения:

+++ https://github.com/cosmos/cosmos-sdk/blob/7d7821b9af132b0f6131640195326aa02b6751db/baseapp/abci.go#L395-L453

`Querier` модуля определяется в файле с именем `querier.go`и состоит из:

- **switch функции** `NewQuerier` для маршрутизации запроса нужной `querier` функции. Эта функция возвращает `querier` функцию, и регистрируется в `AppModule`, чтобы её можно было использовать в менеджере модулей приложения для инициализации маршрутизатора запросов данного приложения - application's query router.

Пример такой switch функции из пособия по службе имён - nameservice tutorial:

```
+++ https://github.com/cosmos/sdk-tutorials/blob/86a27321cf89cc637581762e953d0c07f8c78ece/nameservice/x/n
```

- **Для каждого типа данных модулем определяется одна querier функция, которая должна быть queryable, т.е. ей можно давать запросы**. Разработчики в этих функциях описывают логику обработки запросов, которая обычно включает в себя вызов методов `keeper`-ов для запроса состояния и преобразования (marshalling) его в JSON.

Keeper, кошелёк

Кошельки `Keepers` - это привратники хранилищ в своих модулях. Чтобы записать в хранилище модуля или считать из него, необходимо пройти через один из методов его `keeper`-ов. Это обеспечивается моделью объектных полномочий Cosmos SDK - object-capabilities model. Только объекты, имеющие ключ к хранилищу, могут иметь к нему доступ, и только `keeper` модуля должен хранить ключ(и) к хранилищам модуля.

`Keepers` обычно определяются в файле с именем `keeper.go`. Этот файл содержит определение типа `keeper`-а и методы.

Определение типа `keeper`-а обычно состоит из:

- **Ключ (или ключей)** к хранилищу или хранилищам данного модуля в мультисторе.
- Ссылку на **`keeper`-ы других модулей** - только если самому `keeper`-у нужно иметь доступ к хранилищам других модулей (чтобы читать из них или записывать в них).
- Ссылку на **codec** приложения. Она нужна `keeper`-у для преобразования (marshal) структур перед тем, как их сохранять, или для обратного преобразования их (unmarshal), когда он их извлекает. Потому что хранилища принимают только значения `[]bytes`.

Помимо этого определения типа есть ещё одна важная составляющая файла `keeper.go` - это `NewKeeper`, функция конструктора `keeper`-а. Эта функция создаёт новый инстанс `keeper`-а того типа, который был определён выше, используя в качестве параметров кодек `codec`, ключи `keys`, и, возможно, ссылки на `keeper`-ы других модулей. Функция `NewKeeper`вызывается из конструктора приложения = application's constructor. Остальная часть файла определяет методы `keeper`-а, в первую очередь getter-ы и setter-ы.

Интерфейсы Командной строки (CLI) и REST Интерфейсы

Каждый модуль определяет команды командной строки и пути REST, доступные конечному пользователю через интерфейсы приложения - application's interfaces. Это позволяет конечным пользователям создавать сообщения тех типов, которые были определены в модуле, или запрашивать подмножество состояния, управляемое данным модулем.

CLI - Интерфейсы Командной строки

Как правило, относящиеся к модулю команды определяются в папке с именем `client/cli` в папке модуля. CLI подразделяет команды на две категории, транзакции и запросы, определяемые в `client/cli/tx.go` и `client/cli/query.go` соответственно. И те, и другие команды строятся на библиотеке Cobra - Cobra Library:

- Команды Транзакций позволяют пользователям генерировать новые транзакции, чтобы они могли быть включены в блок, и в результате обновляли состояние. Для каждого типа сообщений в модуле следует создавать одну команду. Эта команда вызывает конструктор сообщения с теми параметрами, которые дал ей конечный пользователь, и оборачивает его в транзакцию. SDK управляет подписыванием и добавлением в транзакцию других метаданных.

- Команды Запросы позволяют пользователям запрашивать подмножество состояния, которое определяется данным модулем. Команды запросов отправляют запросы маршрутизатору запросов данного приложения, и он маршрутизирует их к соответствующему querier-у в зависимости от параметра `queryRoute`.

REST Интерфейс

REST интерфейс модуля позволяют пользователям генерировать транзакции и запрашивать состояние через REST запросы к лёгкому клиентскому демону приложения - light client daemon (LCD). Пути REST определяются в файле `client/rest/rest.go`, который состоит из:

- Функции `RegisterRoutes`, которая регистрирует каждый путь, определённый в этом файле. Эта функция вызывается из главного интерфейса приложения для каждого модуля, используемог в приложении. В SDK используется маршрутизатор Gorilla's mux.

- Определений своих (кастомных) типов запросов для каждой функции создания запроса или транзакции, которые надо предоставить. Эти кастомные типы запросов строятся на базе типа `request` из Cosmos SDK:

  +++ https://github.com/cosmos/cosmos-sdk/blob/7d7821b9af132b0f6131640195326aa02b6751db/types/rest/rest.go#L47-L60

- По одной функции обработчика для каждого запроса, который может маршрутизироваться в данный модуль. Эти функции реализуют основную логику, которая должна обслуживать запрос.

Интерфейс Приложения

Интерфейсы позволяют конечным пользователям взаимодействовать с полно-нодовыми клиентам. Это означает запрос данных от полной ноды или создание и отправку новых транзакций, чтобы они были оттранслированы полной нодой и их в итоге включили в блок.

Главный интерфейс - это Интерфейсы Командной строки (CLI). CLI приложения SDK строится на основе всех CLI команд, определённых в каждом модуле, используемом данным приложением. CLI приложения обычно имеет суффикс `-cli` (например, `appcli`) и определяется в файле `cmd/appcli/main.go`. Этот файл содержит:

- **Функцию** `main()`, которая выполняется для того, чтобы создать `appcli` интерфейс клиента. Эта функция подготавливает каждую команду и добавляет её к `rootCmd` перед тем, как их выполнить. В корне `appCli` функция добавляет общие (generic) команды вроде `status`, `keys` и `config`, команды запросов, команды tx и `rest-server`.

- **Команды запросов** добавляются с помощью вызовов функции `queryCmd`, которая также определяется в `appcli/main.go`. Эта функция возвращает Cobra команду, которая содержит в себе команды запросов, определённые в каждом из модулей приложения. (Они передаются как массив из `sdk.ModuleClients` из функции `main()`). А также некоторые другие низкоуровневые команды, такие, как запросы блока или валидатора. Команды запросов вызываются с помощью CLI команды `appcli query [query]`.

- **Команды транзакций** добавляются с помощью вызвова функции `txCmd`. Аналогично `queryCmd` данная функция возвращает Cobra команду, содержащую tx команды, определённые в каждом из модулей приложения, а также более низкоуровневые tx команды, вроде подписывания транзакции или её распространения (broadcasting).

defined in each of the application's modules, as well as lower level tx commands like transaction signing or broadcasting. Tx команды вызываются с помощью CLI команды `appcli tx [tx]`.

- **Функцию** `registerRoutes`, которая вызывается из функции `main()` при инициализацииlёгкого клиентского демона данного приложения - light-client daemon (LCD) (т.е рест-сревера, `rest-server`). Функция `registerRoutes` вызывает функцию `RegisterRoutes`каждого из модулей приложения, регистрируя тем самым пути модуля на маршрутизаторе демона LCD. LCD можно стартовать запустив команду `appcli rest-server`.

См. пример главного CLI файла приложения из пособия по службе имён - nameservice tutorial

+++ https://github.com/cosmos/sdk-tutorials/blob/86a27321cf89cc637581762e953d0c07f8c78ece/nameservice/cmd/nscli/main.go

Зависимости and Makefile

Это дополнительный, опциональный параграф, так как разработчики свободны в выборе менеджера зависимостей и способа сборки проекта. Скажем лишь, что самый на данный момент популярный фреймворк для управления версиями - это `go.mod`. Он обеспечивает импорт корректной версии каждой библиотеки, использованной в приложении. См. пример из nameservice tutorial:

+++ https://github.com/cosmos/sdk-tutorials/blob/c6754a1e313eb1ed973c5c91dcc606f2fd288811/go.mod#L1-L18

Для сборки приложения обычно используют Makefile. Makefile в первую очередь обеспечивает, что `go.mod` запускатеся до сборку двух входных точек (entrypoints) приложения, `appd` и `appcli`.

См. пример из nameservice tutorial:

+++ https://github.com/cosmos/sdk-tutorials/blob/86a27321cf89cc637581762e953d0c07f8c78ece/nameservice/Makefile

Далее

Узнайте больше о том, как устроен Жизненный Цикл Транзакции {hide}