Overview

We have a 3 tier application separated in Frontend (CLI), Middleware and Backend. Here you can find all documentation for the differents tiers.

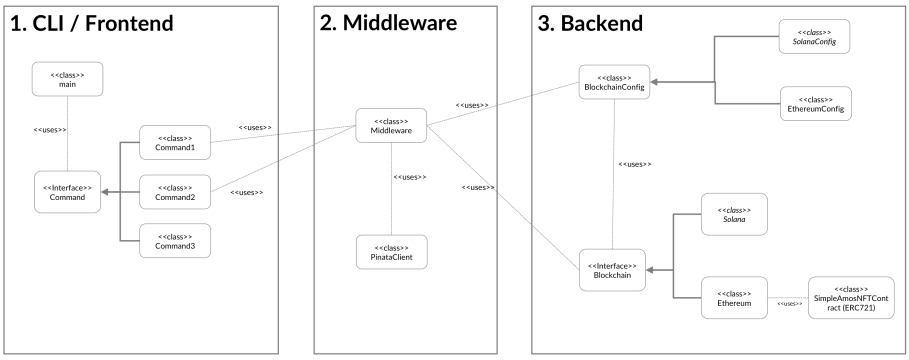

Figure 1: designoverview

CLI

The CLI is implemented using the **inquirer** library. For coloring terminal outputs **chalk** is used.

Every button in a menu is realized by implementing the **Command** interface. The elements of a menu are stored in an array of type `Command[]`. The order of elements in the Array equals the displayed order. HelpCommand and BackCommand are a bit special. Please read the documentation for those commands if you want to use them.

Menus are displayed in a loop. This loop must be broken to exit the menu. The **BackCommand** can be used for this.

See **Command.ts** for the implementation of the main menu.

All commands displayed in the main menu are implemented within the **Commands** folder.

Command Interface

- `name` The label displayed on the button
- `help` The help text that is automatically displayed by a parent menu's help command.
- `execute()` All of a command's functionality is implemented here. This can be either purely functional code or a submenu.

Help Command

Must be instantiated and then a `Command[]` must be stored in `commandIndex`. For an example please see **Command.ts**

- `name` Behaves as described in Command Interface
- `help` Currently ignored
- `execute()` prints the help texts of all commands stored in `commandIndex`

Back Command

This is the only command that needs some parameters `(handler: {run: boolean}, name: string, help: string)`.

See **Command.ts** for an usage example.

- `handler` An object that contains a boolean named `run`. This boolean is used as the loop condition in the parent menu. The back command sets the boolean to false in order to break the loop and thus exit the menu.
- `name` (optional) Behaves as described in Command Interface. Defaults to **Back**. In the main menu, **Exit** is set as the name.
- `help` (optional) Sets the help text.

CliStrings.ts

Contains all strings that are visible to the user. For example button labels, help texts, prompts etc.. Static texts are just plain strings. Texts that have to be formatted at runtime have to be implemented as functions.

Middleware

The middleware is a layer that is situated in between the frontend and backend. On the one hand it's use is to abstract from the backend, so that the frontend does not have to cope with the backend's complexity and on the other hand it enables the frontend to store the users inputs at a central location, the so called `SettingsData.ts`.

When the frontend wants to call a backend functionality it can do so by using the corresponding interfaces provided by the middleware. The middleware will then retrieve all necessary information from its central datastructure. This data is then used to execute the blockchain specific operation on the backend. If there is a result, it is passed back to the frontend when necessary.

Another functionality provided by the middleware, is the `PinataClient.ts`. It enables the user to easily upload files to the **I**nter **P**lanetary **F**ile **S**ystem IPFS. To guarantee the persistent Pinning/Accessibility of the image, through IPFS, we use the so called Pinata Service.

SettingsData.ts

The middleware uses a central datastructure which is composed of a dict of SettingsData-Objects that are mapped to a specific blockchain. Each SettingsData-Object stores a multitude of information for a blockchain like for example **user_private_key**, **blockchain_entry_point** , etc..

The frontend is able to store those values by using **setter-functions** provided by this middleware. When the frontend needs to access those values (for example for validation purposes), it can do so by using corresponding **getter-functions**.

PinataClient.ts

This service can be addressed by the frontend using the following function: **upload_image()**. This function receives the local file path to an image as a parameter and creates a filestream, which is then passed to the pinataSDK. This SDK then uses **API_KEYS**, provided by the user, to authenticate at the pinata endpoint. After authentification the image is uploaded and pinned by pinata so that it is stored persistently and be accessed through the global IPFS network.

Backend

Overview

This is an overview for the backend components. Grey is already implemented, white is tbd in upcoming sprints (latest documented version is from Sprint 6).

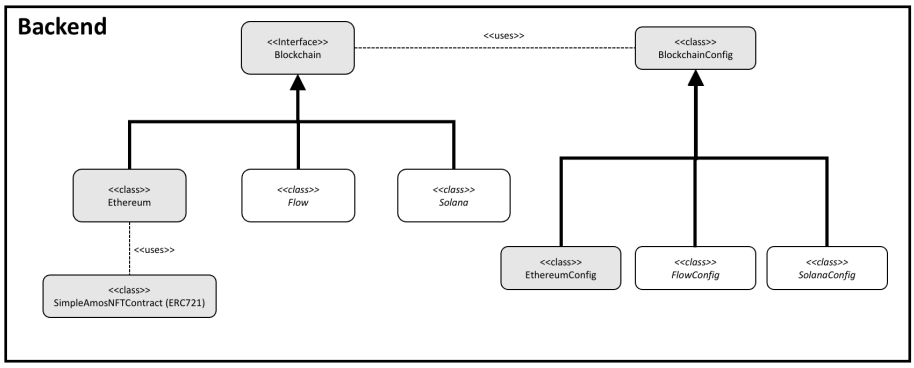

Figure 2: Components_Backend

Generic Blockchain interface

The backend provides a consistent way to connect to different blockchains via a single interface. The backend provides one single interface (`Blockchain.ts`),

which can be used to access multiple Blockchains the same way. Methods like

- `deploy_contract(BlockchainConfig config)`
- `estimate_gas_fee_mint(BlockchainConfig config)`
- `mint_nft(BlockchainConfig config)`
- `read_smart_contract(BlockchainConfig config)`
- `read_user_data_from_smart_contract(BlockchainConfig config)`
- `read_pic_data_from_smart_contract(BlockchainConfig config)`

are offered in a generic way.

Generic Blockchain Config

Every method of the listed one's above needs to runtime a config parameter from a dedicated Blockchain-Type (e.g. `EthereumConfig` for minting an NFT to Ethereum using `mint_nft(EthereumConfig)`.

Dedicated Ethereum implementation

In order to provide functionality for contract and NFT handling on Ethereum a dedicated class `Ethereum.ts` implements `Blockchain.ts`. In this class, all blockchain specific details are encapsulated. Every Method runs asynchronously and holds powerful error handling. The middleware is used to hide all implementation details from the user (in that case the cli), which leads to a clean and scaling architecture.

Dedicated Solana implementation

In order to provide functionality for program accounts and NFT handling on Solana a dedicated class `Solana.ts` implements `Blockchain.ts`. This class encapsulates all blockchain specific details. Every Method runs asynchronously.

Dedicated Flow implementation (still an issue)

FCL (Flow Client Library) is a Typescript library for the Flow blockchain provided by onfolow.org The API-Reference can be found under: https://docs.onflow.org/fcl/reference/api/ The caveat is, that the implementation of the FCL API is designed to be run in a browser => when run, FCL starts an GUI-Overlay on top of the website, which is used to authenticate the user. This Overlay is necessary since there is no other supported method for authentication. The most significant problem we encountered while trying to implement Flow, considering the previously described aspects of FCL the most logical approach, was in our opinion to run a webserver providing a website which runs the FCL and communicates with the backend (CLI) using HTTP-Requests and the webserver mentioned before.

When implementing the plan, we encountered a massive problem which crushed all visions:

There is no simple JS file which could have been imported in the HTML-Document since onflow does not provide such.

To overcome this problem, we tried multiple methods of packing the JS-library into a bundle (Bundle-js, Webpack, ...). Sadly, none of these tools were able to generate an executable JS bundle.

For that reason, we could not come up with an obvious way to enable interaction between FCL, which requires to be run in a browser, and our CLI.

The only possibility would have been to build a react/angular application, which itself manages the frontend dependencies and therefore resolves our problem.

We briefly tried to use the FCL with angular and it seemed to work. But due to the advanced stage of our project, the already existing complexity, and the uncertainty about how well the communication between Web-App and Backend would work (especially in respect to the cryptographically signed keys) we decided to not dive too deep into the implementation of an additional Web-App.

(Note for future projects: It would have been very helpful to start with an angular/react application, since most of the APIs and interactions are designed for a web app running in a browser.)

Process of extending the project with more Blockchains

The architecture of the backend enables us to easily expand the project to new Blockchains, just by adding a new driver that is capable of handling that Blockchain (aka implement the 'Blockchain.ts' interface and providing a corresponding Config-class. Considering the agile nature of this project, the so gained expandability might become very useful later on.