wildework Added warning for Windows users. ... 3 days ago History

..	
artist	8 days ago
hello	8 days ago
readMe.md	3 days ago
sample.flow.json	8 days ago

readMe.md

Fast Floward | Week 1 | Day 3

Welcome to Day 3 of Fast Floward! Today we go back to the always useful `flow-cli` to host the entire Flow environment locally. We'll get to create accounts, sign transactions, and deploy contracts. All of that with the help of the **Flow Emulator**, which is a tiny version of the full Flow blockchain on your computer. Now that's awesome!

Before we get to that, let's review day 2.

Day 2 Review

- Flow Playground is an online environment that simulates the Flow blockchain.
  - You can deploy contracts, send transactions, and execute scripts.
- We store our decentralized applications using Cadence `contract` objects.
- Cadence `script` 's allow us to read data from the Flow blockchain.
- Cadence `transaction` 's allow us to interact with the Flow blockchain.

    - Read and write to account storage using `AuthAccount` during the `prepare` phase.
    - Execute transaction logic in the `execute` phase.
    - Create new instances of Cadence contract `resource` objects.
- Cadence **Account Storage API** and two of its methods
    - `account.save(T, to: /storage/path)`
        - Save objects to account storage.
    - `account.link(/[public|private]/path, target: /domain/path)`
        - Create capabilities that allow borrowing references to objects in storage.
- References give us a way to interact with resources and other types without copy/move operations.
    - Create by casting `&something as &T`.
    - Borrow from capability `account.getCapability().borrow()`.

For a complete Cadence language reference, please visit docs.onflow.org.

Videos

- Day 2 Review, Flow Emulator, and flow.json
- Deploying contracts, sending transactions, and executing scripts

Office Hours

- Quest implementation and general Q&A

Flow Emulator

We saw how the Flow blockchain works in practice on day 2. With Flow Playground we were able to deploy smart contracts, run scripts, and execute transactions. It's awesome that you can do all that in your browser!

To accomplish this, Flow Playground abstracts away most of the details. With **Flow Emulator**, we'll get to run a lightweight version of the Flow blockchain locally. To get started, let's create our first project configuration file.

Configuration

Open up your terminal and run this command.

```
flow init
```

It will create a configuration file, `flow.json`, in the current directory. Its main purpose is to link up your local environment with aliases for contracts and accounts, as well as organizing your contract deployments. Let's explore some of the sections in `flow.json`.

networks

Here is where you will find the addresses of the main Flow blockchain environments. When you want to execute a script, or send a transaction to a particular environment, just use one of the names under `networks`, I.e. `emulator`, `testnet`, or `mainnet`.

On your journey to becoming a DApp developer, look at these environments as a ladder you climb towards shipping a production ready DApp. First, start with the `emulator`. Then, once you a working smart contract, ship it to `testnet` to start integrating with wallets and testing it with friends. Finally, after you've exhaustively tested your DApp, ship it to the `mainnet`.

```
"networks": {
  "emulator": "127.0.0.1:3569",
  "mainnet": "access.mainnet.nodes.onflow.org:9000",
  "testnet": "access.devnet.nodes.onflow.org:9000"
}
```

accounts

Here you will assign aliases to the accounts you're using in the development process. Every `flow.json` starts by having `emulator-account` defined, otherwise known as a *service account*. You can add other accounts to simulate various scenarios that your DApp can encounter, or for contract deployment and many other purposes.

```
"accounts": {
  "emulator-account": {
    "address": "f8d6e0586b0a20c7",
    "key": "bcbd7e16179f286eeb805e06482ac45657d1dface4a775511abcaf8e4b6d4373"
  }
}
```

`contracts`

While using Flow Playground we got to interact with smart contracts by `import`'ing them from accounts using the account address. However, if you hard-code an account address in your Cadence code, you will only be able to deploy that contract in the environment that has that account. This is because account addresses are not shared between the emulator and the testnet, or mainnet.

To solve for this, simply use `import Contract from "path/to/contract.cdc"` in your *.cdc* files, and the `flow-cli` will take care of replacing that with the appropriate account address, if one is defined in your `flow.json`.

```
"contracts": {
  "Hello": "./hello/contract.cdc"
}
```

`deployments`

We combine all of the above sections to create a mapping for contract deployment: **network** > **account** > **contract**.

```
"deployments": {
  "emulator": {
    "emulator-account": [
      "Hello"
    ]
  }
}
```

How to Lauch

If you get a system dialog asking for network permissions, please allow it.

```
flow emulator start
```

You should be greeted with 4 `INFO` messages, with two at the end confirming the start of a `gRPC` and a `HTTP` server, indicating that you're good to go. The emulator will continue running as long as you keep the process running. You can stop it using the `SIGINT` terminal signal (`CTRL + C` on a macOS terminal).

The data you create while running your emulator doesn't persist by default. This is useful when running tests. However, if you want to, you can instruct the emulator to persist its data using the `--persist` flag.

```
flow emulator start --persist
```

This will create a `flowdb` folder for the emulator to store its state. Of course, you can always delete the `flowdb` folder to reset your state.

Playground

Now that we know how the `flow.json` configuration file works and how to start the Flow Emulator, let's start using it. With the `flow emulator` process running, open up another tab in your terminal where we can start running our commands. By the way, the following commands work for all Flow environments, not just the emulator. The emulator is just the default, if you don't explicitly set a different `--network` .

Keys

We want to create an account. Before we can do that, we'll need a key pair. Rememeber, the blockchain uses cryptography to determine ownership. The `flow-cli` has a handy command to do just that. The command below also sets the signature algorithm to `ECDSA_secp256k1` , which is the same signature algorithm that Bitcoin uses.

```
flow keys generate\
  --sig-algo=ECDSA_secp256k1
```

This returns a set of public and private keys. Obviously, you should not make them public, so be careful.

```
Private Key
70d4eebade37eabe0a5df1b1664acf25245187068665c529c1d63f0a214dadfa
Public Key
c69560acb6ff5b4db1870ec47c6f2474f862b34bb69b3508557e5733406da63cb5218bdf4ddeb
```

Accounts

With our keys in hand, we can create our account. Please note that we must identify the signature algorithm if it's not the default `ECDSA_P256`. Also, since creating an account is just a regular transaction under the hood, an account must sign it. Thankfully, the emulator comes prepopulated with `emulator-account`.

```
flow accounts create \
  --key "c69560acb6ff5b4db1870ec47c6f2474f862b34bb69b3508557e5733406da63cb521
  --sig-algo "ECDSA_secp256k1" \
  --signer "emulator-account"
```

If the command succeeds, you'll get an account address in your response. Quick note, unlike Ethereum and some other chains, the account address is not derived from the public key. This means that simply having a public key is not enough to identify an account.

```
Transaction ID:
71b9ded371f66716170d012d2962d97b3dd5c8d820cb62ef775c770949220953

Address  0x01cf0e2f2f715450
Balance  0.00100000
Keys     1
```

To confirm it all worked, you can get account information by running the following command (please replace the account id with the one you get assigned).

```
flow accounts get 0x01cf0e2f2f715450
```

Finally, we can update our `flow.json` by adding this newly generated account to it. We're going to use the long-form `key` definition.

```
{
  ...,
  "accounts": {
    ...,
    "emulator-artist": {
                        "address": "01cf0e2f2f715450",
                        "key": {
                                "type": "hex",
                                "index": 0,
                                "signatureAlgorithm": "ECDSA_secp256k1",
                                "hashAlgorithm": "SHA3_256",
                                "privateKey": "70d4eebade37eabe0a5df1b1664acf
```

```
                    }
                }
    },
    ...
}
```

Please refer to the docs for more details.

Contracts

Great! So we have an account, now let's use it to deploy our `Hello` contract that's located in the `hello` folder. Notice how we're no longer using `log()` for our greeting. That's because logging only works on Flow Playground and in the Cadence REPL shell. Instead, we're going to return the greeting as a string.

Another change we're making - we're adding an `event`. When interacting with a contract, it's useful to have events as a way to communicate when certain things happen. Transactions don't have return statements and don't print logs in the emulator, so using events will allow us to see our `greeting`. For more on events, please refer to the docs.

```
pub contract Hello {
  pub event IssuedGreeting(greeting: String)

  pub fun sayHi(to name: String): String {
    let greeting = "Hi, ".concat(name)

    emit IssuedGreeting(greeting: greeting)

    return greeting
  }
}
```

First, let's update our `flow.json` by giving this contract a string name, and providing a path to the source code.

```
{
  ...,
  "contracts": {
    "Hello": "./hello/contract.cdc"
  },
  ...
}
```

Then we define contract deployment targets.

```
{
  ...,
  "deployments": {
    "emulator": {
      "emulator-artist": [
        "Hello"
      ]
    }
  }
  ...
}
```

At this point, we can go ahead and deploy our project.

```
flow project deploy
```

If we did everything right, we should see this output.

```
Deploying 1 contracts for accounts: emulator-artist

Hello -> 0x01cf0e2f2f715450
(bed8b44ec08dace72775e07a89ceb2ae24949b8ba8991da824bd0895b10ef36e)
```

Please refer to the docs for more details.

Scripts

Congrats! We now have our first real smart contract deployed with an account. Let's waste no time, and go play with it. The easiest way to do this will be to execute a script.

Same as with our updated `Hello` contract, we'll modify the script from day 2 to return a string, instead of logging it. Also, we'll provide a `name: String` as an argument instead of hard-coding it. It's more fun that way! One caveat before we proceed, we have to comment out the `emit event` from our `sayHi` function because scripts don't support events.

```
import Hello from "./contract.cdc"

pub fun main(name: String): String {
```

```
    return Hello.sayHi(to: name)
}
```

To execute we'll use this command. Please note how we supply arguments by using `--arg Type:value` flags.

```
flow scripts execute hello/sayHi.script.cdc \
  --arg String:"FastFloward"
```

Warning! If you're using Windows PowerShell or any other non-unix like terminal, please only use `--arg` to set your arguments. `--args-json` doesn't work on Windows.

We can also use **JSON** to encode our arguments. Remember, the top-level has to be an array `[]`, like in this example.

```
flow scripts execute hello/sayHi.script.cdc \
  --args-json='[{"type": "String", "value": "FastFloward"}]'
```

Please refer to the docs for more details on scripts, as well as the how to encode Cadence values as JSON.

Transactions

Read-only access to the Flow blockchain with scripts is cool, but transactions are ice cold! Let's take a look.

Flow transactions go through a pipeline that starts with the Cadence source code.

1. Build: encode the source code using `rlp` or the "Recursive Length Prefix" encoding.
2. Sign: cryptographically sign the encoded source code.
3. Send: deliver the encoded source code with the signatures to a Flow Access Node.

For more, please refer to the docs.

Right, time to build! Here's a sample command to build our `sayHi.transaction.cdc` transaction.

```
flow transactions build ./hello/sayHi.transaction.cdc \
  --authorizer emulator-artist \
  --proposer emulator-artist \
  --payer emulator-artist \
```

```
  --filter payload \
  --save transaction.build.rlp
```

The result is a file `transaction.build.rlp` that encodes our source code, as well as the necessary payer, proposer, and authorizer information. We can then proceed to signing it with our `emulator-artist` account.

```
flow transactions sign ./transaction.build.rlp \
  --signer emulator-artist \
  --filter payload \
  --save transaction.signed.rlp \
  -y
```

This produces another file `transaction.signed.rlp` and we can finally send it to our emulator for processing.

```
flow transactions send-signed ./transaction.signed.rlp
```

If all goes well, as part of the response, we should see our event that we defined earlier.

```
Events:
    Index       0
    Type        A.01cf0e2f2f715450.Hello.IssuedGreeting
    Tx ID
2188e78921960e1f4cb336432159c8a161f84a0a336bf3afb33a3f77e0ce7f5e
    Values
                - greeting (String): "Hi, 0x1cf0e2f2f715450"
```

Shortcut

We went through each step of the transaction pipeline to gain a better understanding of how it works. Going forward, we should use the shortcut version of this process.

```
flow transactions send ./hello/sayHi.transaction.cdc \
  --signer emulator-artist
```

Quests

We're getting closer and closer to the our goal of shipping decentralized applications! Today's quests will get just a step away from the finish line. Let's go get it! You'll find stubs for these quests in the `/artist` folder.

- `W1Q5` – Event calendar

Update your `Artist` contract from day 2 to include 3 new events. Emit those events when you see fit.

```
pub event PicturePrintSuccess(pixels: String)
pub event PicturePrintFailure(pixels: String)
```

- `W1Q6` – Printer goes brrrrr

Implement the following transactions.

```
createCollection.transactions.cdc
print.transaction.cdc
```

- `W1Q7` – What you got?

Implement the `displayCollection.script.cdc` as per the specification in the file.

Best of luck on your quests!