Lesson 6

Solana Development

Resources
Documentation
Solana Cookbook

Solana Concepts

See. Terminology

Major points

- In Solana everything is an account
- All programs are stateless
- Programs are executable accounts
- Accounts are used to store data
- Each account has a unique address
- Accounts have a max size of 10MB

Account

Accounts have only one owner and only the owner may debit the account and adjust its data.
Executable accounts (programs) are owned by the BPF Loader.
An account is essentially a file with an address (public key) that lives on the Solana blockchain. If you need to store data on chain it gets stored in an account.
An account must pay rent in order to persist on chain.

Programs

On Solana smart contracts are called programs. A program is just an account that has been marked executable.
Once a program has been deployed on chain it can be read and interacted with via instructions.

Program address

This the account associated with the program account that holds the program's data (shared object).

Program ID

The public key of the account containing a program. This address can be referenced in an instruction's program_id field when invoking a program.

Program Derived Addresses (PDAs)

PDAs enable programs to create accounts that only the program can modify.

BPFLoader

The Solana program that owns and loads BPF smart contract programs, allowing the program to interface with the runtime.

Sealevel

The Solana network runtime that enables the parallel execution of instructions and transactions. Though very different in ,implementation and modelling, it is equivalent to the Ethereum Virtual

Machine.

Native Programs

"Special" programs that are required to run the network.

- System Program: The system program is responsible for creating new accounts, and assigning account ownership.
- BPF Loader: The BPF Loader program is responsible for deployment, upgrades and instruction execution of Solana programs.
- There are other native programs but these are the more relevant programs for Solana programming.

Token Programs

A kind of program that implements a fungible or non fungible token other than the native $SOL token.

Associated Token Accounts

If an account holds any token other than the native Solana token it will have an associated token account for each type of token it holds.

Instructions

What is called in order to execute a function of a program.

Sysvar

An account which enforces certain variables of the network such as epoch, rent, validator rewards, etc.

Rent

The network charges rent (in $SOL) for data held in accounts, since this takes up validator network resources. An account can be exempt from rent collection if it has 2 years of rent in it's balance. Rent is collected every epoch. If an account is unable to rent it will no longer load.

Accounts in more detail

See Account Model
and Cookbook

Solana separates code from data, all programs are stateless so any data they need must be passed in from the outside.
All accounts are owned by programs.
Some accounts are owned by System program, and some can be owned by your own program.

Accounts are both used by and owned by programs, and a single program can own many different accounts.

Account Fields

- key
- isSigner
- isWritable
- lamports
- data
- owner
- executable
- rent_epoch

Owner versus holder

The *owner* is not the person who own the private key of the account ,they are called the *holder*.
The holder is able to transfer the balance from the account.

The owner in has the right to amend the data of any account.
In Solana, system program is set to be the owner of each account by default.
So for example if you create an account in Solana in order to store some SOL you would be the holder but the System program would be the owner.

Difference to Ethereum

On Ethereum, only smart contracts have storage and naturally they have full control over that storage.
On Solana, *any* account can store state but the storage for smart contracts is only used to store the immutable byte code.
On Solana, the state of a smart contract is actually completely stored in other accounts.

- Accounts can store arbitrary kinds of data as well as SOL.
- Accounts also have metadata which describes who is allowed to access its data and how long the account can live for.
- Anyone can read or credit an account, but only the account owner can debit it or modify its data.
- Accounts are created by simply generating a new keypair and registering its public key with the System Program.
- Each account is identified by its unique address, the same as in a wallet.

There are 3 kinds of accounts on Solana:

- Data accounts that store data
- Program accounts that store executable programs
- Native accounts that indicate native programs on Solana such as System, Stake, and Vote

Within data accounts, there are 2 types:

- System owned accounts
- PDA (Program Derived Address) accounts

Every account in Sealevel has a specified owner.
Since accounts can be created by simply receiving lamports, each account must be assigned a default owner when created.
The default owner in Sealevel is called the “System Program”.
The System Program is primarily responsible for account creation and lamport transfers.

Program Derived Address (PDA)

A Program Derived Address is an account that’s designed to be controlled by a specific program.
With PDAs, programs can programatically sign for certain addresses without needing a private key.
At the same time, PDAs ensure that no external user could also generate a valid signature for the same address.

It may be helpful to consider that PDAs are not technically `created`, but rather `found`.
We will go into the details of this in future lessons.

Development Tools

More on these tomorrow

Solana playground

See playground

Local Validator

See Docs
You need to have installed the solana-cli, see lessons 3 and 4
Run
`solana-test-validator` to start it.

Anchor Framework

We will look at this next week, once we have the covered the basics of programs on Solana.

Rust continued

Using modules within your programs

You can bring modules into your program with the
`use` keyword .
For example
`use std::env;`

Error handling

Rust distinguishes between recoverable and un recoverable errors, we will handle these errors differently

1. Recoverable errors
   With these we will probably want to notify the user, but there is a clear structured way to proceed. We handle these using the type `Result<T, E>`
   `enum Result<T, E> { Ok(T), Err(E), }`
   Many standard operations will return a type of `Result` to allow for an error
   A good pattern to handle this is to use matching

```
        let f = File::open("foo.txt"); // returns a Result
        let f = match f {
                Ok(file) => file,
                Err(error) => // some error. handling
                }
```

You can propagate the error back up the stack if that is appropriate

```
let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
};
```

THE ? OPERATOR

A shortcut for propagating errors is to use the ? operator

```
let mut f = File::open("hello.txt")?;
```

The `?` placed after a `Result` value works in almost the same way as the `match` expressions.
If the value of the `Result` is an `Ok`, the value inside the `Ok` will get returned from this expression, and the program will continue.
If the value is an `Err`, the `Err` will be returned from the whole function as if we had used the `return` keyword so the error value gets propagated to the calling code.

2. Un recoverable errors
   In this case, we have probably come across a bug in the code and there is no safe way to proceed. We handle these with the `panic!` macro
   For a discussion of when to use `panic!` see the docs
   When the `panic!` macro executes, your program will print an error message, unwind and clean up the stack, and then quit.
   We can specify the error message to be produced as follows

```
panic!("Bug found ....");
```

Cargo

Cargo is a tool used to compile and build projects and manage dependencies
(the alternative is to use rustc - the rust compiler)
For example we can use cargo to set up a project

```
cargo new --bin bootcamp
```

will create a rust package in a `bootcamp` directory, that will result in an executable (rather than a library)

It creates a default `main.rs` file for you :

```
fn main() {

println!("Hello, world!");

}
```

Metadata is stored in the Cargo.toml file

```
[package]
name = "bootcamp"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
```

To build and run the project we created we use
`cargo run`

COMMON CARGO COMMANDS ARE

```
    build, b    Compile the current package
    check, c    Analyze the current package and report errors, but don't build
object files
    clean       Remove the target directory
    doc, d      Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    run, r      Run a binary or example of the local package
```

```
test, t     Run the tests
bench       Run the benchmarks
update      Update dependencies listed in Cargo.lock
search      Search registry for crates
publish     Package and upload this package to the registry
install     Install a Rust binary. Default location is $HOME/.cargo/bin
uninstall   Uninstall a Rust binary
```

Packages, crates and modules

A `crate` is a binary or library, a. number of these (plus other resources) can form. a `package` , which will contain a *Cargo.toml* file that describes how to build those crates.
A crate is meant to group together some functionality in a way that can be easily shared with other projects.
A package can contain at most one library crate, but. any number of binary crates.
There can be further refinement with the use of `modules` which organise code within a crate and can specify the privacy (public or private) of the code.
Module code is private by default, but you can make definitions public by adding the `pub` keyword.

In summary a package can contain multiple crates, and each crate can contain multiple modules.