Lesson 9

This week

Mon : Program development continued
Tues : Introduction to Web3
Weds : Anchor Introduction
Thurs : Anchor continued

Staking

Staking through a staking pool

Steps

1. Install the stake pool API
   `cargo install spl-stake-pool-cli`

2. Use the standard commands to get the config, set the cluster and keypair

   `solana config get`

   `solana config set --url https://api.devnet.solana.com`

   `solana config set --keypair ~/my-solana-wallet/new-keypair.json`

3. Create a stake pool
   `spl-stake-pool create-pool --epoch-fee-numerator 3 --epoch-fee-denominator 100 --max-validators 1000`

4. Deposit into the pool
   `$ spl-stake-pool deposit-sol <Pool ID> <amount>`

See the documentation for further config and commands

SPL Governance

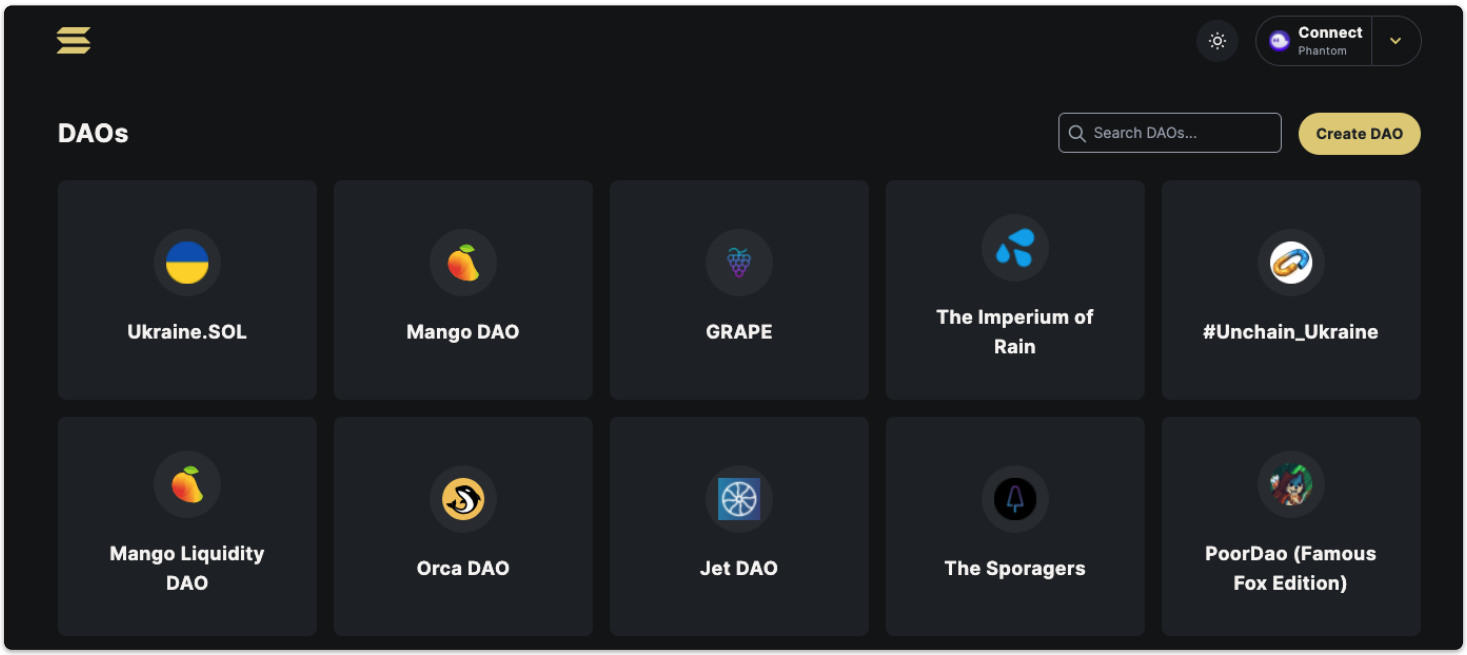
See docs
and UI docs

The program can be used to build any type of DAOs which can own and manage any type of assets.

For example it can be used as an authority provider for mints, token accounts and other forms of access control where we may want a voting population to vote on disbursement of funds collectively. It can also control upgrades of itself and other programs through democratic means.

In the simplest form the program can be used for Multisig control over a shared wallet (treasury account) or as a Multisig upgrade authority for Solana programs

DAO Wizard



You can create a

- Multisig DAO
- Bespoke DAO
- NFT based DAO

Further aspects of Solana Programs

Compute Budget

From Documentation

To prevent a program from abusing computation resources, each instruction in a transaction is given a compute budget. The budget consists of computation units that are consumed as the program performs various operations and bounds that the program may not exceed. When the program consumes its entire budget or exceeds a bound, then the runtime halts the program and returns an error.

The following operations incur a compute cost:

- Executing BPF instructions
- Calling system calls
  - logging
  - creating program addresses
  - cross-program invocations

Each transaction roughly has the fixed cost and it naturally puts pressure on developers to optimize on-chain code to fit within the system limits. Transactions do have fees on Solana, though.

You may come across :

```
Program GJqD99MTrSmQLN753x5ynkHdVGPrRGp35WqNnkXL3j1C consumed 200000 of 200000
compute units
Program GJqD99MTrSmQLN753x5ynkHdVGPrRGp35WqNnkXL3j1C BPF VM error: exceeded
maximum number of instructions allowed (193200)
```

Currently for both mainnet and testnet，the limit of computation units is 200K，for log it is 100.

You may also see:

```
Program log: Error: memory allocation failed, out of memory
```

It means that you run out of memory, the limit for the stack is 4kb and 32kb for the heap.
You should be really careful when you use structures that can potentially use a large amount of heap, such as Vec or Box.
When coding for the blockchain you need to pay more attention to memory than in traditional technology.

PS: Pay attention to recursive functions calls, the depth should not exceed 20.

Logging the compute budget from Rust

Use the system call `sol_log_compute_units()` to log a message containing the remaining number of compute units the program may consume before execution is halted

---

Breaking program into modules

Modules give code structure by introducing a hierarchy similar to the file tree. Each module has a different purpose, and functionality can be restricted.
At the root module multiple modules are compiled into a unit called a crate.
Crate is synonymous with a 'library' or 'package' in other languages.

Modules are defined using the `mod` keyword and often contained in `lib.rs`.
A Common pattern seen throughout program implementations is:

```
// lib.rs
pub mod entrypoint;
pub mod error;
pub mod instruction;
pub mod processor;
pub mod state;
```

Where:

- `lib.rs`: registering modules
- `entrypoint.rs`: entrypoint to the program
- `instruction.rs`: (de)serialisation of instruction data
- `processor.rs`: program logic
- `state.rs`: (de)serialisation of accounts' state
- `error.rs`: program specific errors

Programming Model

See Docs
An app interacts with a Solana cluster by sending it transactions with one or more instructions.
The Solana runtime passes those instructions to programs deployed by app developers beforehand.
An instruction might, for example, tell a program to transfer lamports from one account to another or create an interactive contract that governs how lamports are transferred. Instructions are executed sequentially and atomically for each transaction.
If any instruction is invalid, all account changes in the transaction are discarded.

Sealevel — Parallel Processing of smart contracts

From Introduction

On Solana, each instruction tells the VM which accounts it wants to read and write ahead of time. This is the root of the optimisations to the VM.

1. Sort millions of pending transactions.
2. Schedule all the non-overlapping transactions in parallel.

(Aside see Ethereum new transaction type and access list)

SIGNATURE CHECKING

The parallel nature of Sealevel means that signatures can be checked quickly using GPUs.

Transaction structure

Transactions specify a collection of instructions
Each instruction contains the program, program instruction, and a list of accounts the transaction wants to read and write.

- Signatures — this is a list of ed25519 curve signatures of the message hash, where the "message" is made up of metadata and "instructions".
- Metadata, the message has a header, which includes 3 fields describing how many accounts will sign the payload, how many won't, and how many are read-only.
- Instructions, this contains three main pieces of information:
  - a) set of accounts being used and whether each one is a signer and/or writable,
  - b) a program ID which references the location of the code you will be calling
  - c) a buffer with some data in it, which functions as calldata.

Accounts

On Solana blockchain everything is an account and they are pages in the shared memory.

OVERVIEW

Account type	Executable	Writable
Program	✅	❌
Data keypair owned	❌	✅
Data program owned (PDA)	❌	✅

Accounts maintain:

- arbitrary data that persists beyond the lifetime of a program
- balance in SOL proportional to the storage size
- metadata relating to permissions

Accounts have an "owner" field which is the Public Key of the program that governs the state transitions for the account.
Programs are accounts which store executable byte code and have no state. They rely on the data vector in the Accounts assigned to them for state transitions. All programs are owned by BPF Loader (BPFLoaderUpgradeab1e11111111111111111111111).

A PDA has no private key.

1. Programs can only change the data of accounts they own.
2. Programs can only debit accounts they own.
3. Any program can credit any account.
4. Any program can read any account.

By default, all accounts start as owned by the System Program.

1. System Program is the only program that can assign account ownership.
2. System Program is the only program that can allocate zero-initialized data.
3. Assignment of account ownership can only occur once in the lifetime of an account.

A user-defined program is loaded by the loader program. The loader program is able to mark the data in the accounts as executable. The user performs the following transactions to load a custom program:

1. Create a new public key.
2. Transfer coin to the key.
3. Tell System Program to allocate memory.
4. Tell System Program to assign the account to the Loader.
5. Upload the bytecode into the memory in pieces.
6. Tell Loader program to mark the memory as executable.

At this point, the loader verifies the bytecode, and the account to which the bytecode is loaded into can be used as an executable program. New Accounts can be marked as owned by the user-defined program.

The key insight here is that programs are code, and within our key-value store, there exists some subset of keys that the program and only that program has write access.

Msg Macro

See docs
msg! can be used to output text to the console in Solana

Rust

Traits examples in Solana

A trait tells the Rust compiler about the functionality of a generic type and defines shared behaviour. Traits are often called interfaces in other languages, although with some differences.

A `trait` is a collection of methods defined for an unknown type: `Self`. Trait methods can access other trait methods.

```
trait Details {
    fn get_owner(&self) -> &Pubkey;
    fn get_admin(&self) -> &Pubkey;
    fn set_owner(&self) -> &Pubkey;
    fn set_admin(&self) -> &Pubkey;
    fn get_amount(&self) -> u64;
    fn set_amount(&self);
    fn print_details(&self) {
            println!("Owner is {:?}", self.get_owner())
            println!("Admin is {:?}", self.set_admin())
            println!("Amount is {}", self.get_amount())
    }
}
```

If certain methods are frequently reused between structs, it makes sense to define a trait.

Implementations

Implementations are defined with the `impl` keyword and contain functions that belong to an instance of a type, statically, or to an instance that is being implemented.

For a struct `AccountA` representing the layout of data on chain:

```

pub struct AccountA {
        pub admin: Pubkey,
        pub owner: Pubkey,
        pub amount: u64,
}
```

To implement previously defined abstract functions of the trait `Details`:

```
impl Details for AccountA {
        fn get_owner(&self) -> &Pubkey {
                        &self.owner
        }
        fn get_admin(&self) -> &Pubkey {
                        &self.admin
        }
        ...
}
```

In Solana this is often used for instructions relating to serialisation of the data stored under a given account:

```
impl AccountA {
        fn unpack(input: &[u8]) -> Result<Self, ProgramError> {
                ...
        }

        fn pack(&self, dst: &mut [u8]) {
                ...
        }
}
```

Then in the processor bare bytes can be converted to a usable struct as simply as

```
let mut account_temp = AccountA::unpack(ADDRESS)?;
```

Errors review

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 ....");
```

Practical Instructions

Prerequisites

The examples require the following items to work:

- Rust compiler / Cargo manager
- Solana client
- Solana work repo

Cloning the example repo

```
git clone https://github.com/ExtropyIO/SolanaBootcampOctober
```

If you prefer you can use gitpod
`gitpod.io/#/https://github.com/ExtropyIO/SolanaBootcampOctober

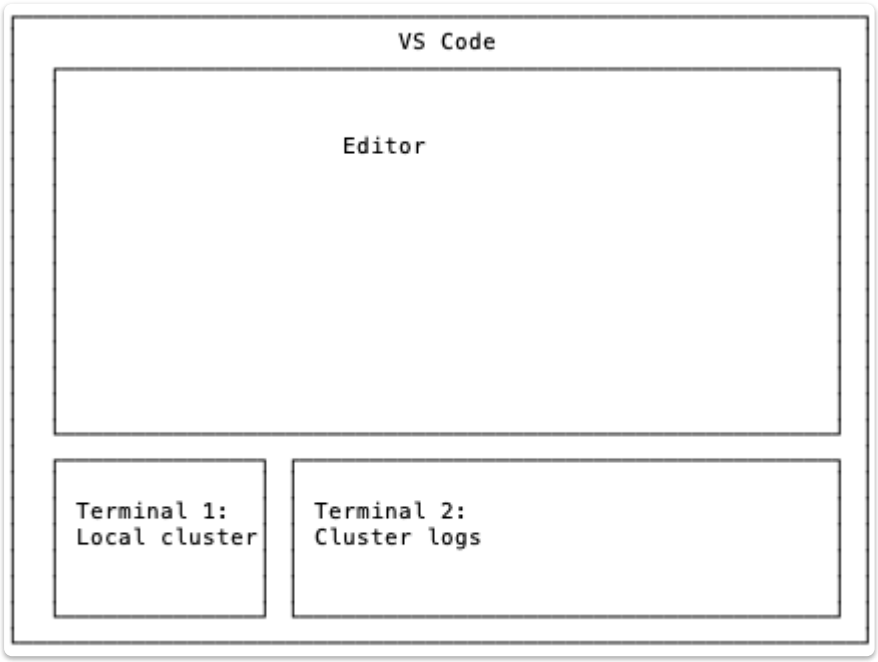
Set-up for development

Windows

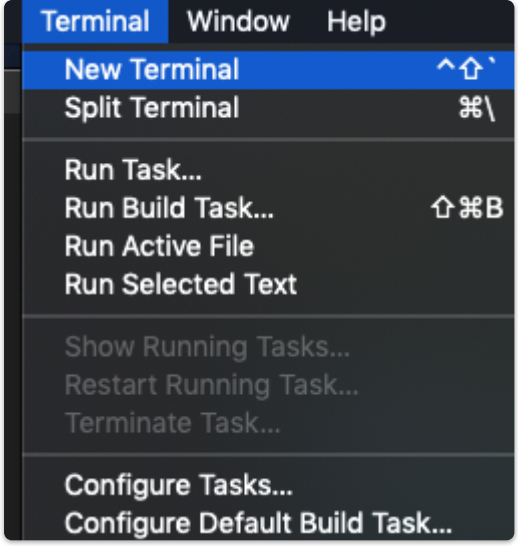
Local development set-up requires a terminal instance for:

- Local cluster
- Cluster logs

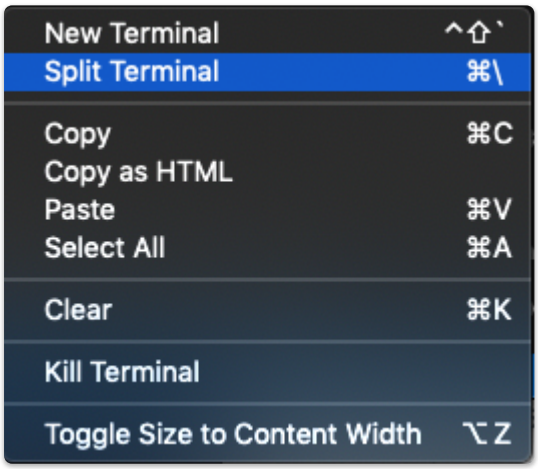
It is quite easy to set this up within a single VS code session as such:



To open a new terminal instance select:



To split it into two right click on the terminal window and select:



TMUX can be also be used if prefered.

Starting a local cluster

To start a local validator run `solana-test-validator`.
Doing so will create an output of the following format:

```
--faucet-sol argument ignored, ledger already exists
Ledger location: test-ledger
Log: test-ledger/validator.log
⠁ Initializing...
Identity: 3qrx65hH5NmogVYrYMaNA93BjJVuWDH39qJTkT8HzM49
Genesis Hash: BniBn8Fzv6TYe59g4yR1ie6XsjLUvxfkPz9EX8YQfXgY
Version: 1.9.17
Shred Version: 21208
Gossip Address: 127.0.0.1:1024
TPU Address: 127.0.0.1:1027
JSON RPC URL: http://127.0.0.1:8899
⠉ 00:00:37 | Processed Slot: 321619 | Confirmed Slot: 321619 | Finalized Slot: 3
```

From this we can see that local RPC server is listening on port 8899 on the local network. Requests from client will be to that IP and that URL.

Setting the right network

To check the network that the client will connect to run
`solana config get`.

The following settings will be output:

```
Config File: <USER>/.config/solana/cli/config.yml
RPC URL: https://api.devnet.solana.com
WebSocket URL: wss://api.devnet.solana.com/ (computed)
Keypair Path: <USER>/.config/solana/id.json
Commitment: confirmed
```

On the second line it is displayed that the client will connect to the devnet.

Running
`solana config set --url http://localhost:8899`
will set the default network to localhost.
Running
`solana config get`
again gives:

```
Config File: /Users/chavka/.config/solana/cli/config.yml
RPC URL: http://localhost:8899
WebSocket URL: ws://localhost:8900/ (computed)
Keypair Path: /Users/chavka/.config/solana/id.json
Commitment: confirmed
```

With that set, tests will know which cluster to connect to in order to interact with our program.

Open cluster logs

In the second terminal window run
`solana logs`

This window will display information from the cluster such as logs relating to:

- deployment
- execution costs
- cross program invocations
- custom logs

Messages are communicated from the program using the `msg!` macro.

Getting enough balance

The balance for a given account can be retrieved with
`solana balance`

To top up your wallet (the one who's private key is referenced here
`Keypair Path: <USER>/.config/solana/id.json`)
run
`solana airdrop <SOL_AMOUNT>`

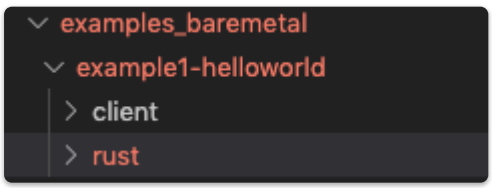
This doesn't always work as expected, especially on devnet and the chance of succesful airdrop is higher with SOL balances being no more than two, so run
`solana airdrop 2`.
Tests can be written in such way to automatically airdrop SOL to the accounts that will require them.

Repo overview

The purpose of the repo 'solana-course' is to provide examples which work right out of the box and allow any potential developer to immerse themselves in the Solana development envriroment by exposure to tools as well as examples of programs, tests and clients.

Each example in the `examples_baremetal` directory has:

- rust code that will be deployed to the cluster in the `/rust` directory
- typescript code that will interact with on-chain program in the `/client` directory



Additionaly in the main root there is a file called `utils.ts` which has wrappers for frequently reused functions such as `establishConnection()` or `loadKeypair()`.

Baremetal in this instance refers to interaction with the hardware without applications or an operating system at a very low level.
This is in contrast to Anchor (which we will cover later) that abstracts away quite a lot of the tedious boilerplate code and in process greatly enhances development speed.

Commands for these examples are contained in the `package.json` file at the repo root directory.

Compilation

Examples provided can be compiled with
`npm run build`
which will compile all of the programs at once.
Under the hood it will invoke modified
`cargo build-bpf`, `cargo build-bpf --workspace --manifest-path=./examples_baremetal/Cargo.toml`
which will build binaries for all the examples in the baremetal directory.

Deployment

In the example directory to deploy a given program command run
`npm run deploy:X`
where X is the number of the example program to deploy.
This has to be done from within the same directory where `package.json` is.

Interaction

To go along with the example programs there are short, clear and well documented clients written in typescript that allow connection to the pre-deployed contracts.

To run a client run
`npm run call:X`
where X is the number of the example program to interact with.
This has to be done from within the same directory where `package.json` is.

Client functions

In essence each client executes the following steps upon each invocation:

1. Load libraries
2. Load binary
3. Load binary keypair
4. Load payer keypair
5. Establish connection to the cluster (specified in client config)
6. Build serialised representation of arguments and function to invoke
7. Call the program
8. Read modified on-chain state
9. Log cost of interacting with the program

Additional checks exist such as those that check whether caller has enough SOL or whether program in question has been deployed. These not being satisfied will log a comment which will allow the user to fix the problem.

For the examples in the repo

See repo
In the project root run

`npm i`

Example 1

1. Compile
   `npm run build`
2. Deploy
   `npm run deploy:1`
3. Interact
   `npm run call:1`

This program receives no instruction data, no accounts and it only logs a message using the `msg!` macro that can be viewed in the window running `solana logs`

EXTRA LINKS:

Solana Cookbook - Accounts
Solana Docs - Accounts
Solana Wiki - Account model
