## Lesson 8 - Solana development

### 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 err

---

### Program derived addresses

See article

See wiki

Solana design constraints:

- All state has to be fed to the program
- A program can only alter the state of the accounts that it owns

Program can modify any account that it owns, it just can't sign for it.


Program derived accounts enable programs to create accounts that the program can sign for.

Being able to sign means you can open as well as to close an account.

Account Signing Authority

1. Solana accounts can only be assigned to a program if the account's signing authority approves the change. Typically, the signing authority just means that the corresponding private key must sign the transaction.
2. Since program execution state is entirely public and known to every validator, there's no way for it to secretly sign a message to create an account. To allow

account creation by programs, the Sealevel runtime provides a syscall which allows a program to derive an address from its own address which the program can freely claim to sign.

Program Derived Addresses (PDAs) designed to be controlled by a specific program. With PDAs, programs can programmatically sign for certain addresses without needing a private key. PDAs serve as the foundation for Cross-Program Invocation, which allows Solana apps to be composable with one another.

- PDAs are 32 byte strings that look like public keys, but don't have corresponding private keys
- `findProgramAddress` will deterministically derive a PDA from a programId and seeds (collection of bytes)
- A bump (one byte) is used to push a potential PDA off the ed25519 elliptic curve
- Programs can sign for their PDAs by providing the seeds and bump to invoke_signed
- A PDA can only be signed by the program from which it was derived
- In addition to allowing for programs to sign for different instructions, PDAs also provide a hashmap-like interface for indexing accounts

---

### Example use case

Using a program derived address, a program may be given the authority over an account and later transfer that authority to another. This is possible because the program can act as the signer in the transaction that gives authority.

For example, if two users want to make a wager on the outcome of a game in Solana, they must each transfer their wager's assets to some intermediary that will honour their agreement. Currently, there is no way to implement this intermediary as a program in Solana because the intermediary program cannot transfer the assets to the winner.

This capability is necessary for many DeFi applications since they require assets to be transferred to an escrow agent until some event occurs that determines the new owner.

- Decentralised Exchanges that transfer assets between matching bid and ask orders.
- Auctions that transfer assets to the winner.
- Games or prediction markets that collect and redistribute prizes to the winners.

Program derived address:

1. Allow programs to control specific addresses, called program addresses, in such a way that no external

user can generate valid transactions with signatures for those addresses.

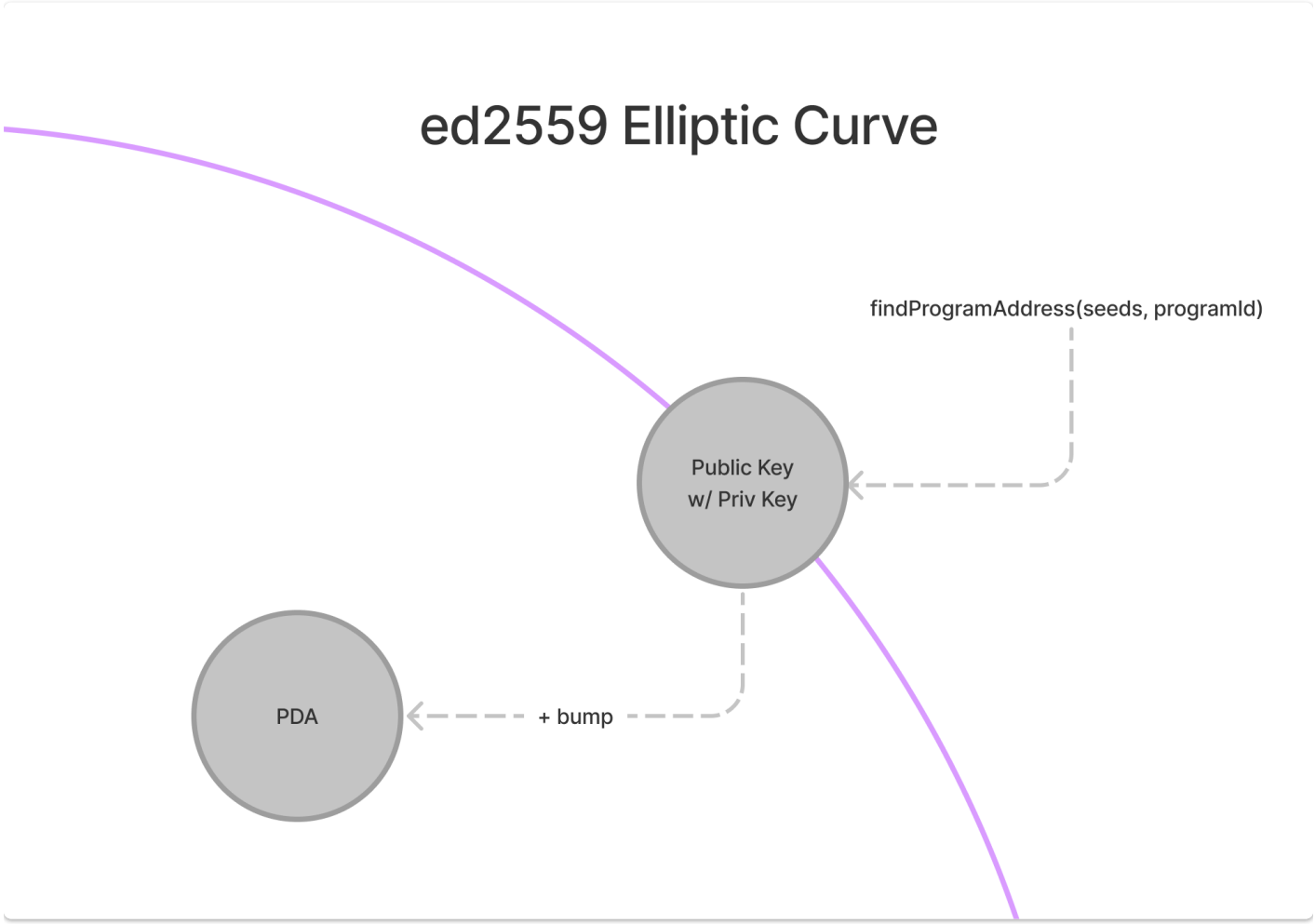
2. Allow programs to programmatically sign for program addresses that are present in instructions invoked via Cross-Program Invocations.

Given the two conditions, users can securely transfer or assign the authority of on-chain assets to program addresses, and the program can then assign that authority elsewhere at its discretion.

---

### PDA derivation details

Program derived addresses are deterministically derived from a collection of seeds and a program id using a 256-bit pre-image resistant hash function.
The program address must not lie on the ed25519 curve to ensure there is no associated private key.
During generation an error will be returned if the address is found to lie on the curve. There is about a 50/50 chance of this happening for a given collection.




Derivation pseudo code:

```
 pda_pubkey =
findProgramDerivedAddress(programId, seeds,
seedBump)
```

For Sealevel to parallelise batches of instructions and to prevent concurrency issues the client needs to provide an account in the list of accounts to be used by the program even if that account doesn't exist yet.

As an account is being created it will need to have some lamports transferred to it to become rent exempt. Depositing lamports will require this account to have its state modified and as such it needs to be marked as writable.

**This means that PDA has to be derived by the client and has to be submitted to the program.**

## Practical Instructions for upcoming homeworks

### 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/SolanaBootcamp
```

If you prefer you can use gitpod or a codespace

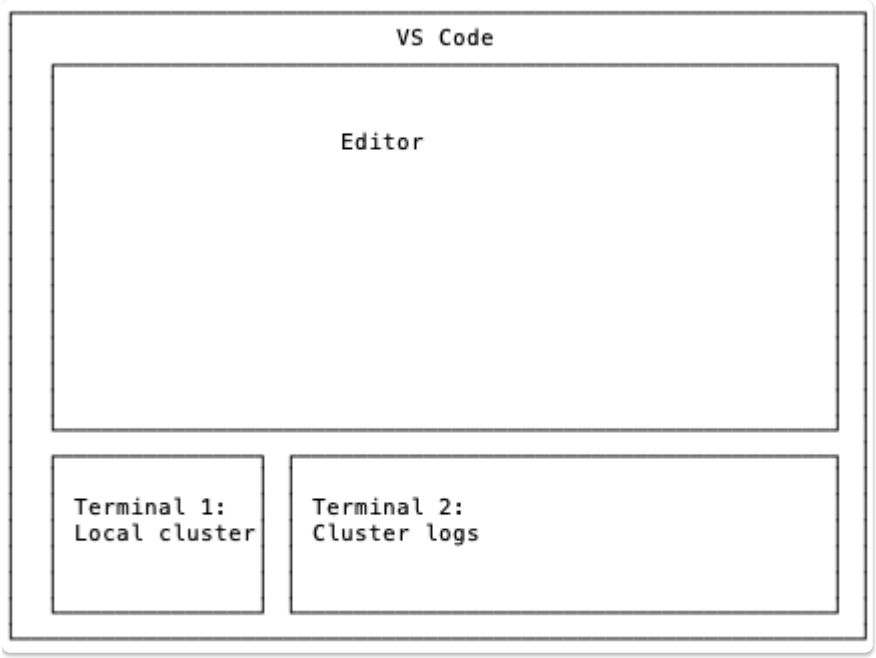
### Set-up for development

#### Windows

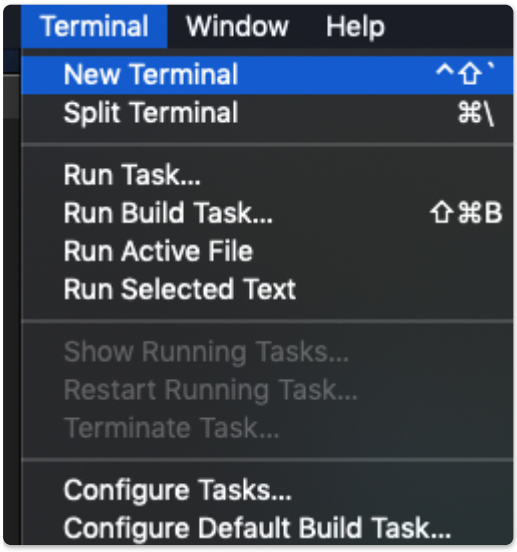
Local development set-up requires a terminal instance for:

- Local cluster
- Cluster logs

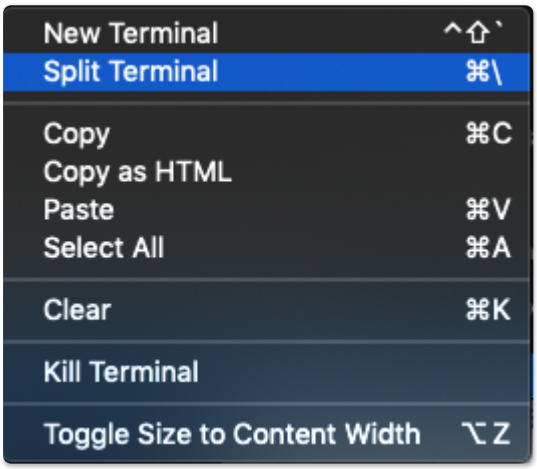
It is quite easy to set this up within a single VS code session as such:




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>`

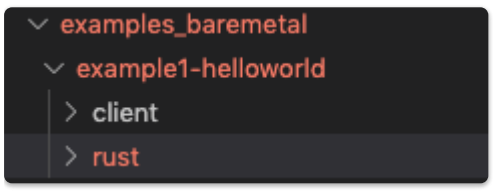
This doesn't always work as expected, especially on devnet and the chance of succesful airdrop is higher with SOL balances being no more than two, so run `solana airdrop 2`.
Tests can be written in such way to automatically airdrop SOL to the accounts that will require them.

### Repo overview

The purpose of the repo 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