Lesson 6

Solana Development

Solana Programs overview

See Docs

- Programs process instructions from both end users and other programs
- All programs are *stateless*: any data they interact with is stored in separate accounts that are passed in via instructions
- Programs themselves are stored in accounts marked as `executable`
- All programs are owned by the BPF Loader and executed by the Solana Runtime
- Developers most commonly write programs in Rust or C++, but can choose any language that targets the LLVM's BPF backend
- All programs have a single entry point where instruction processing takes place (i.e. `process_instruction`); parameters always include:
  - `program_id`: `pubkey`
  - `accounts`: `array`,
  - `instruction_data`: `byte array`

Transactions

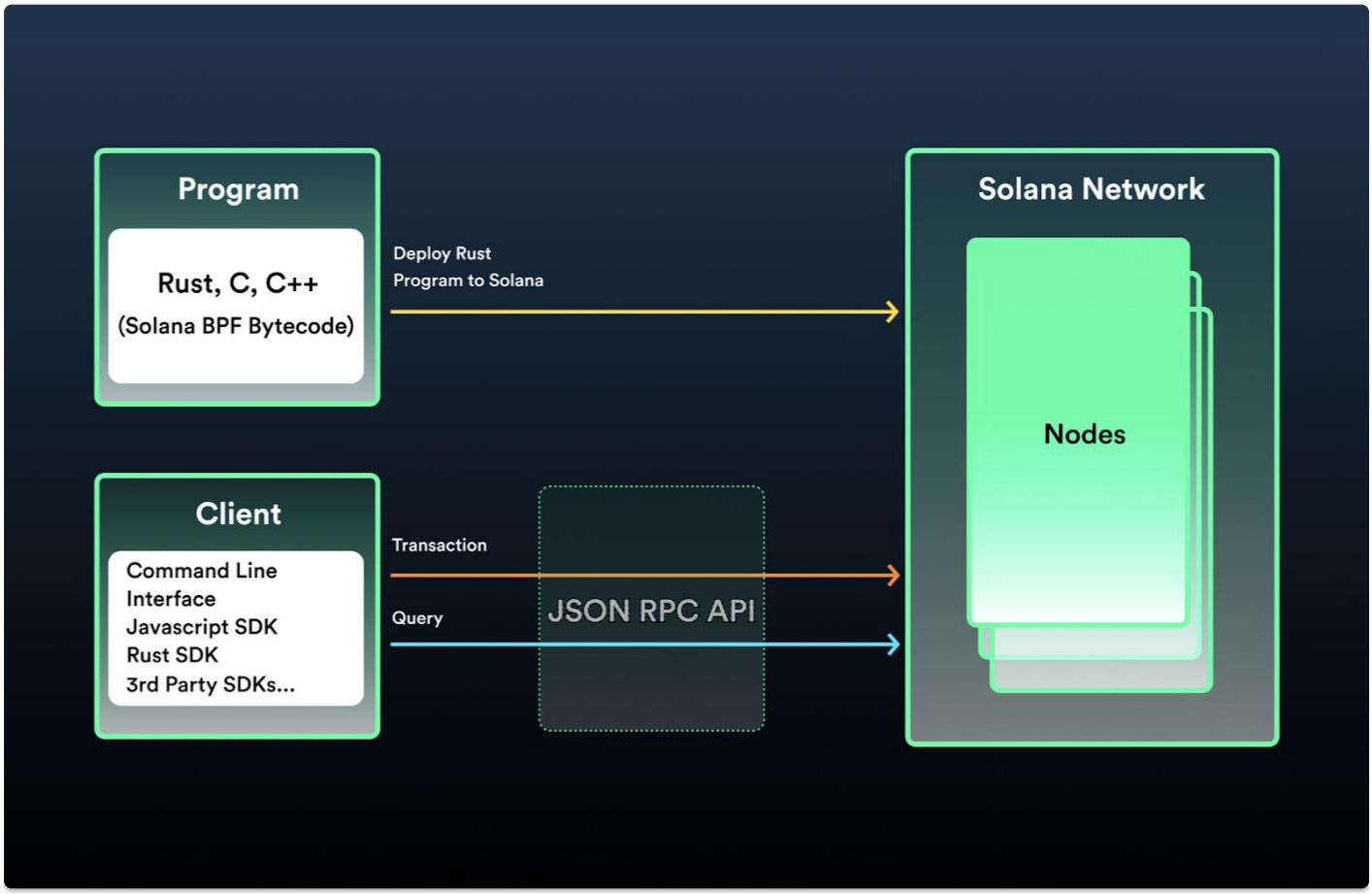
From Cookbook

Clients can invoke programs by submitting a transaction to a cluster. A single transaction can include multiple instructions, each targeting its own program. When a transaction is submitted, the Solana Runtime will process its instructions in order and atomically. If any part of an instruction fails, the entire transaction will fail.

dApp architecture

dApps on Solana have the following parts:

- accounts on Solana chain, which store program binaries and state data
- client that interacts with on-chain accounts using RPC nodes
- additional components such as storage (Arweave/IPFS), task scheduler (Cronos) or input from outside world (Chainlink)

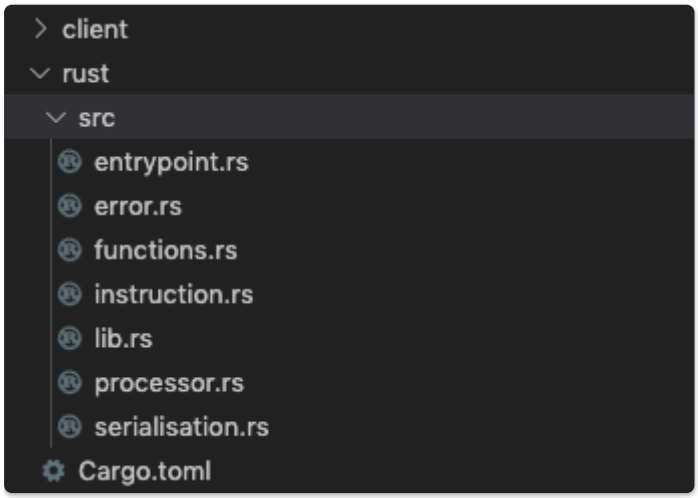


On-chain program architecture

Solana programs (ie a smart contracts) are generally composed of distinct modules, with each module represented by an individual Rust file:

- entrypoint
- instruction
- processor
- state
- error

This is to make reading, maintaining and testing code easier, for smaller projects it is fine to encapsulate total program functionality within a single file. It's up to the designer to break down intended business logic into sensible module layout.



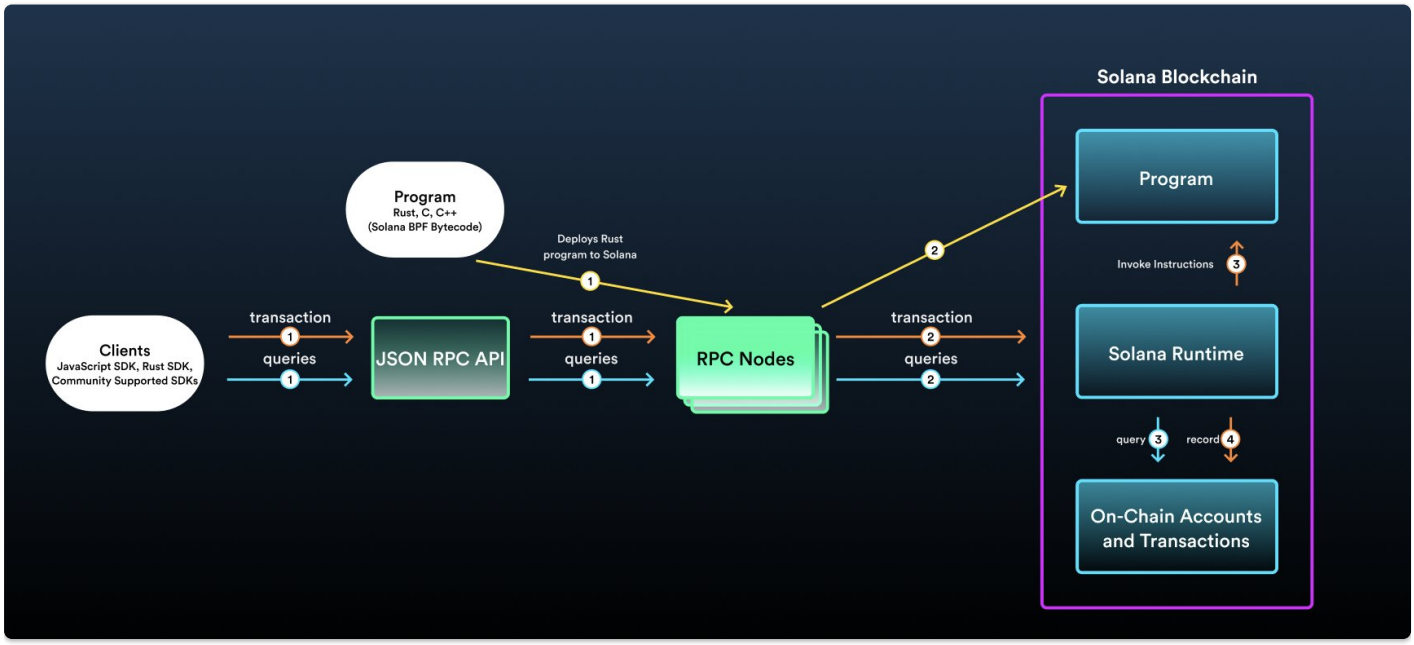
Cargo project architecture

Project set-up will look slightly differently depending on whether Anchor is used or not and whether you are developing the front end or only the smart contracts. Further deviations can come from customisation of the `Cargo.toml` which can be configured in different ways such as where `target` directory is located.

Rust project set-up (including Solana development) generally has the following directories:

- src: logic of the program that will be deployed on chain
- target: binary for deployment and files needed for compilation
- tests: tests for the smart contract
- Cargo.toml: Rust manifest file containting dependencies
- Cargo.lock: autogenerated dependency file

Details about RPC



All client interaction with the Solana network happens through Solana's JSON RPC API.

This means sending JSON object representing a program you want to call, method within that program and arguments to this method which includes list of utilised accounts.

Example of object that can be sent to an RPC node.

```
payload = {
 "jsonrpc": "2.0",
```

```
  "id": 1,
  "method": "getBalance",
  "params": [
    "KEY"
  ]
}
```

- `jsonrpc` - The JSON RPC version number. It needs to be `2.0`
- `id` - An identifier specified for this call, it can be a string or a whole number.
- `method` - The name of the method you want to invoke.
- `params` - An array containing the parameters to use during the method invocation.

Interaction with RPC nodes is achieved using and SDK developed by solana labs.
This library (`@solana/web3.js`) abstracts away significant amount of boilerplate code meaning you can invoke functions rather than having to assemble each time JSON.

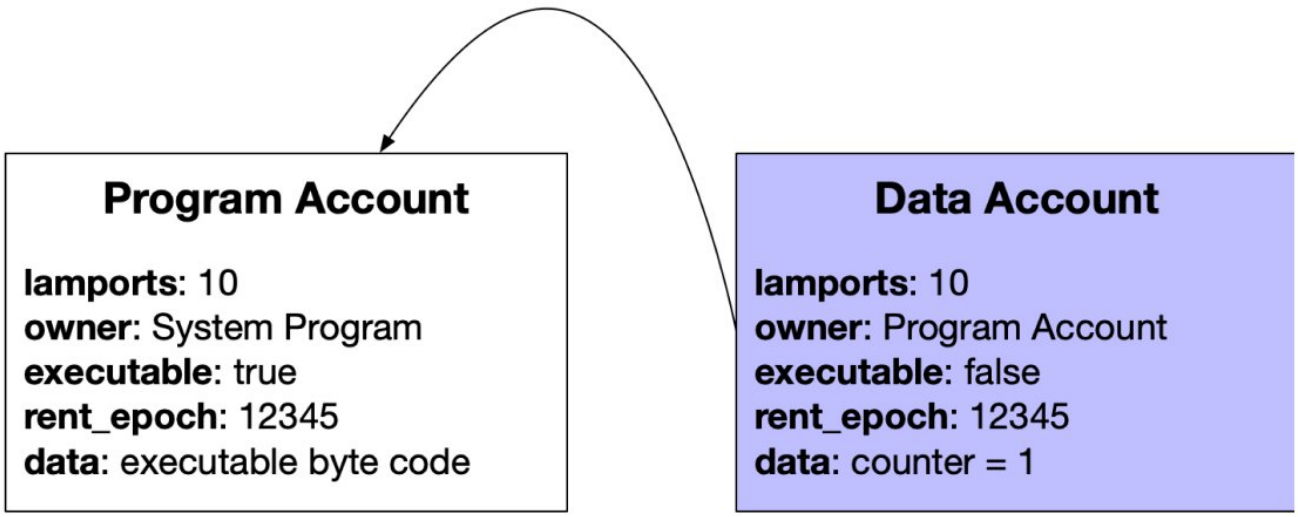
A list of available methods is here but these methods are not the same as the methods within a given program. They are much broader and additional work including serialisation has to be done.

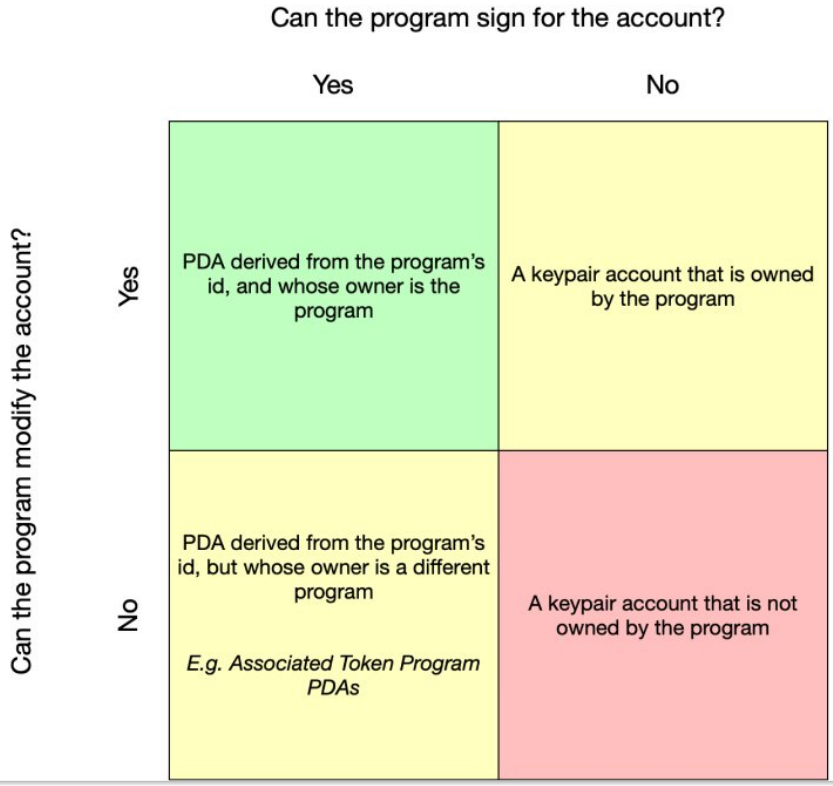
Thankfully there are libraries such as borsh or frameworks such as anchor to make it easier.

Programs in more detail

See Docs

Programs and accounts





Programs and State

Programs are stateless so how do we handle state ?

We need to create other non-executable accounts and set the program as the owner of those accounts.
The program can then the update data of this 'storage' account.

A system account is one that was created by the Solana system program. It is typical that these are often considered a wallet conceptually.
This program is owned by the SystemProgram.

```
Public Key: 8wFGJ5ae5q2nGvcmwSrqxUy8MmHwKMejTV81Bm91RgNw
Balance: 499999997.75602299 SOL
Owner: 11111111111111111111111111111111
Executable: false
Rent Epoch: 0
```

This is an empty account owned by a program.

```
Public Key: 5WBMTK8B3g9b3fkFbS18WRWvxA52MjtDhpVPZF6Ti6zq
Balance: 0.00103008 SOL
Owner: 2pUPsC4tBLePhaX8XbU8hHRUMuZ4MGxhmBHDAWRAfapu
Executable: false
Rent Epoch: 0
Length: 20 (0x14) bytes
0000:   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0010:   00 00 00 00
```

This is an on chain program owned by the BPFLoader.

```
Public Key: 2pUPsC4tBLePhaX8XbU8hHRUMuZ4MGxhmBHDAWRAfapu
Balance: 0.00114144 SOL
Owner: BPFLoaderUpgradeab1e11111111111111111111
Executable: true
Rent Epoch: 0
Length: 36 (0x24) bytes
0000:   02 00 00 00  56 d7 56 a2  e0 d7 62 75  d4 0b f4 5e
0010:   e8 6e b9 ef  9d 30 fc fe  d2 aa 3e f0  d7 a4 eb e6
0020:   14 1f 8c ad
```

Program arguments

Every program has a single entry point and it receives instructions composed of three distinct parts:

- program id
- accounts
- instruction data

```
pub fn process_instruction(
    _program_id: &Pubkey,
    _accounts: &[AccountInfo],
    _instruction_data: &[u8],
) -> ProgramResult {
```

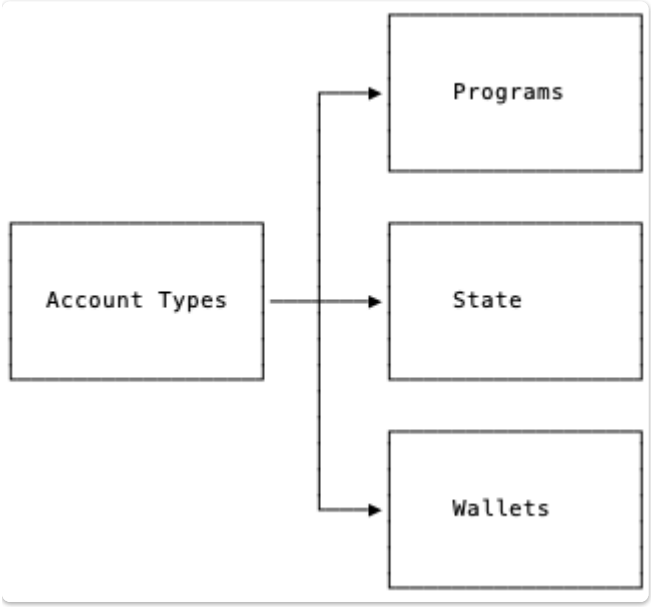
The program can return successfully, or with an error code. An error return causes the entire transaction to fail immediately.
On success you can not return any values, so in addition state has to be checked manually by the client.

Program ID

The instruction's program_id specifies the Public key of the account being invoked. Though program's are statless they can inquire about the ownership of a provided account that it is to attempt interacting with.

Accounts

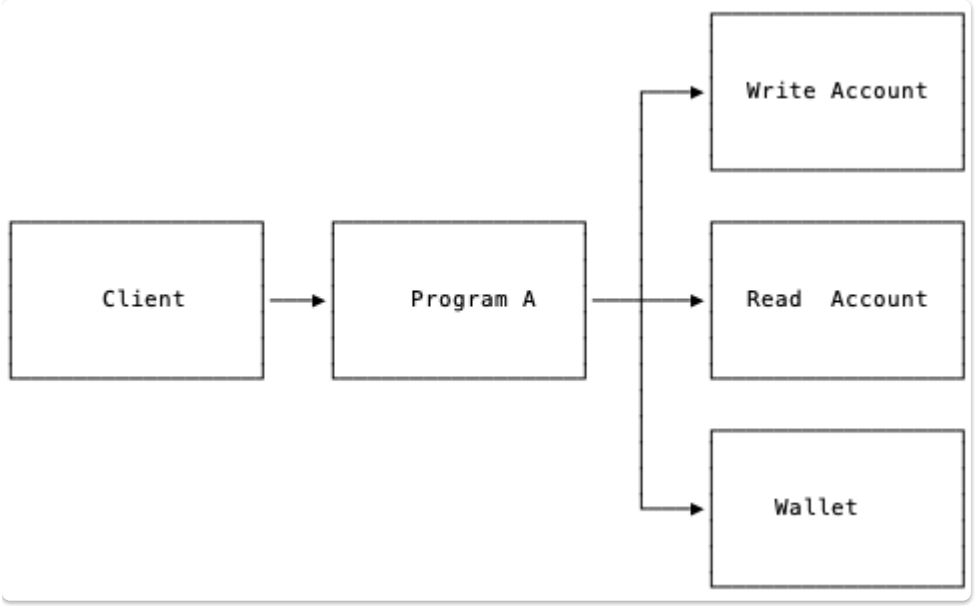
The accounts referenced by an instruction represent all the on chain accounts that this program will interact with and serve as both the inputs and outputs of a program.
Account can be either a program containing logic, data account containing state or users wallet.



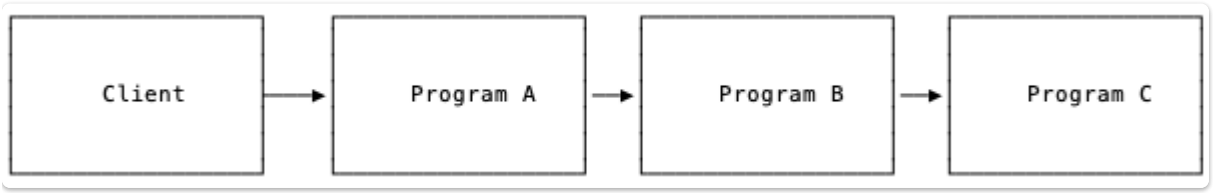
Account passed have to specify whether they will be read only or writeable.

Account is a storage location on Solana blockchain. They store state like the amount of lamports, owner and state or logic data.
Each non-PDA account has a keypair with the public key being the address of that account.

Multiple accounts can be passed as the program might require them to accomplish it logic. It may require to read and modify state of other accounts or to transfer lamports.



Or it may require logic of other programs to supplement its own one.

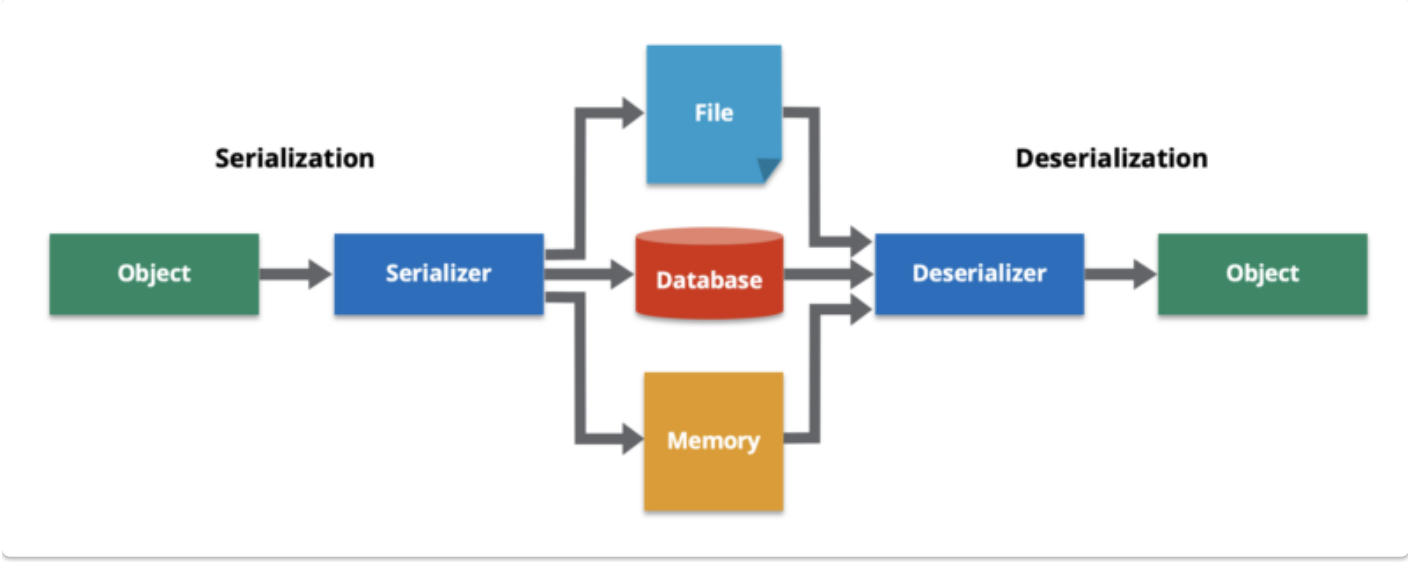


In future lessons we will look at Program Derived Addresses and how we use them when programs want to talk to each other.

Instruction data

Each instruction carries a general purpose byte array that is passed to the program along with the accounts. The contents of the instruction data is program specific and typically used to convey what operations the program should perform, and any additional information those operations may need above and beyond what the accounts contain.

Programs are free to specify how information is encoded into the instruction data byte array. The choice of how data is encoded should consider the overhead of decoding, since that step is performed by the program on-chain. It's been observed that some common encodings (Rust's bincode for example) are very inefficient.



A transaction can contain instructions in any order. This means a malicious user could craft transactions that may pose instructions in an order that the program has not been protected against. Programs should be hardened to

properly and safely handle any possible instruction sequence.

Generic program flow

The basic program flow (excluding RPC call):

1. Serialised arguments (accounts, signatures, instruction) are received by the entrypoint
2. The entrypoint forwards the arguments to the processor module
3. The processor invokes the instruction module to decode the instruction argument
4. Using the decoded data, the processor will now decide which function to use to process this specific request
5. The processor may use the state module to encode state into or decode the state of an account which has been passed into the entrypoint or can be derived programatically
6. If error occurs at any point execution stops and program reverts with general or specific error code

Generic client flow

1. Load Interface Description Language (IDL)
2. Connect to the network
3. Assemble instruction
4. Submit instruction (RPC call)
5. Read modified account state (RPC call)

Development workflow

Developing a program involves iteration over the following steps:

1. Compilation of the Rust code to generate `.so` binary
2. Deployment of the `.so` binary to a cluster
3. Interaction with the program

Then re looping to add, modify, remove or test a given functionality.

Compilation

To compile a program the following command is run:

```
cargo build-bpf
```

`build-bpf` allows the Rust compiler to output Solana compatible Berkley Packet Filter bytecode.

This should be run from the program directory using
`cd <PATH>`
to where there is `Cargo.toml` is located.

On the first compilation it will produce two files into the `/target/deploy directory`:

- `program_name.so` binary that can be deployed to the cluster

- `program_name-keypair.json` private key associated with this program

The name of the `program_name.so` and `program_name-keypair.json` files is set in `Cargo.toml` here:

```
 [lib]
name = "program_name"
```

Deployment

The general format of the command to deploy a program is:

```
solana program deploy <PATH_TO_SO_BINARY>
```

You must have enough lamports in the network that the Solana client is connected to. Scripts can be written to automate deployment of multiple programs.

Interaction

Interacting with the program is dependent on what exactly the client is.
Language and libraries used will differ depending whether the client is a mobile application, a browser plugin or an embedded device.

Programs, State, Data, Rent, Fees

Solana stores only two things in on-chain accounts:

- program binary (and it's hash)

- arbitrary developer specified data

Any user data such as token balances, access rights can be stored in accounts.
Accounts is a bit of a confusing name and files would likely be more accurate. As a developer you chose what each account looks like and what kind of data it stores by defining the serialisation and deserialisation procedure.

We can look at the default example in the playground

Useful Solana Resources

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

Solana Accounts

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`.

Solana Accounts

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`.

Accounts part 2

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

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 (BPFLoaderUpgradeab1e11111111111111111111).

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.

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.

Msg Macro

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

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

Development Tools

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.