Lesson 7 - Solana programs

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

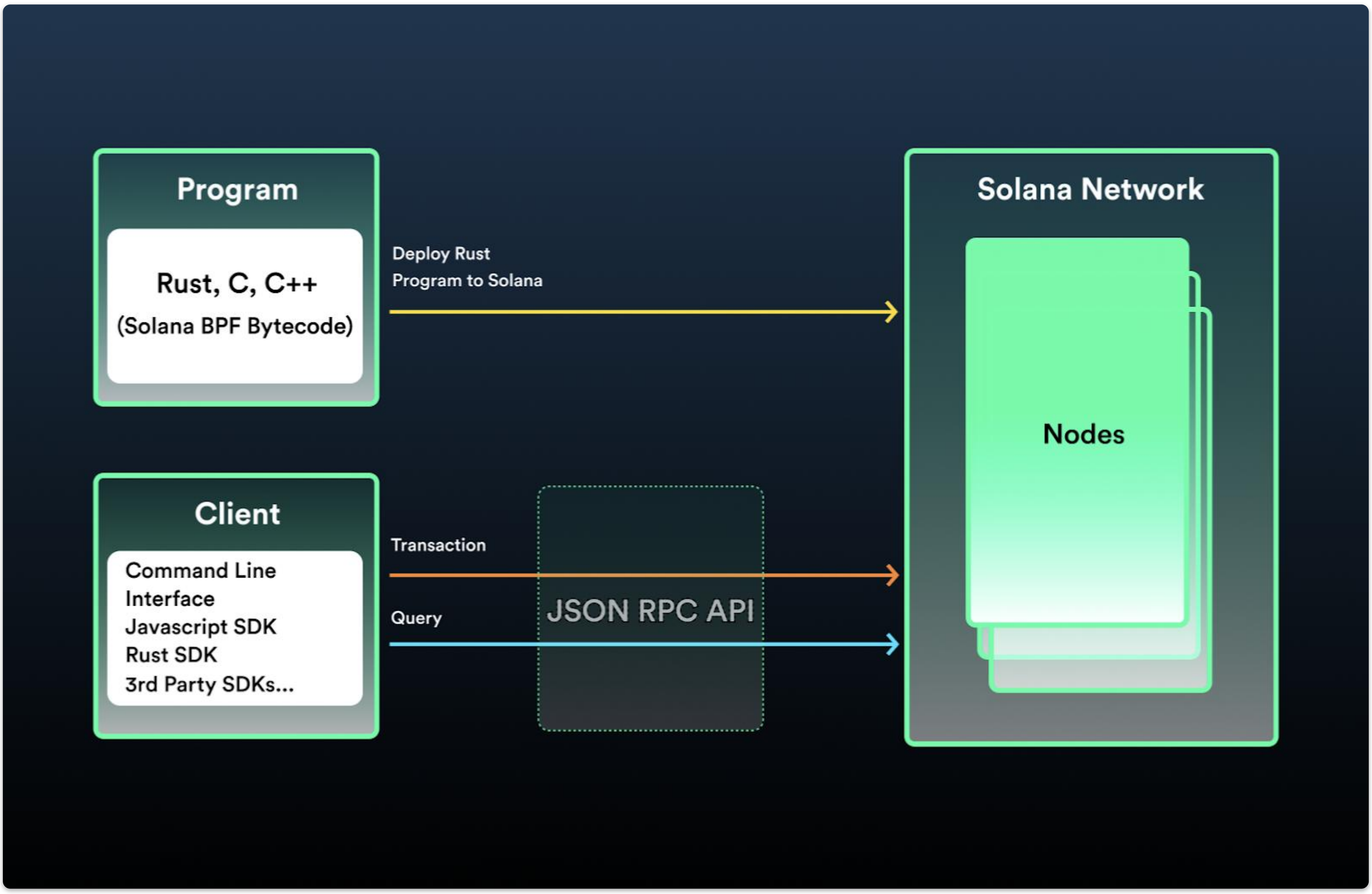
Clients can invoke programs by submitting a transaction to a cluster. A single transaction can include multiple instructions, each targeting its own program. When a transaction is submitted, the Solana Runtime will process its instructions in order and atomically. If any part of an instruction fails,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

---

dApp architecture

dApps on Solana have the following parts:

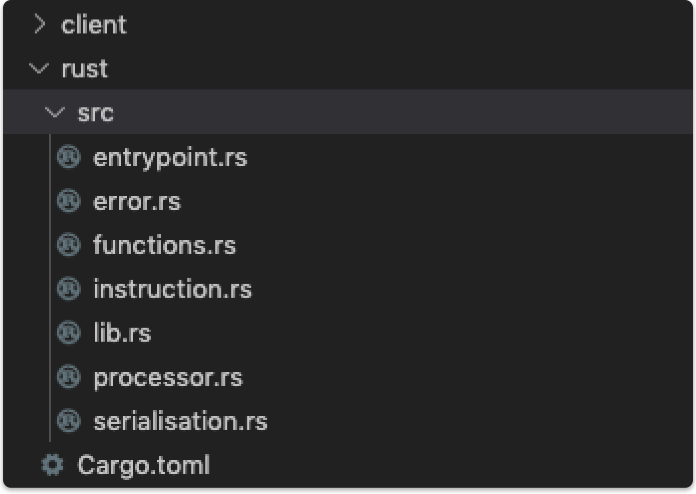
- accounts on Solana chain, which store program binaries and state data
- client that interacts with on-chain accounts using RPC nodes
- additional components such as storage (Arweave/IPFS), task scheduler (Cronos) or input from outside world (Chainlink)



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.

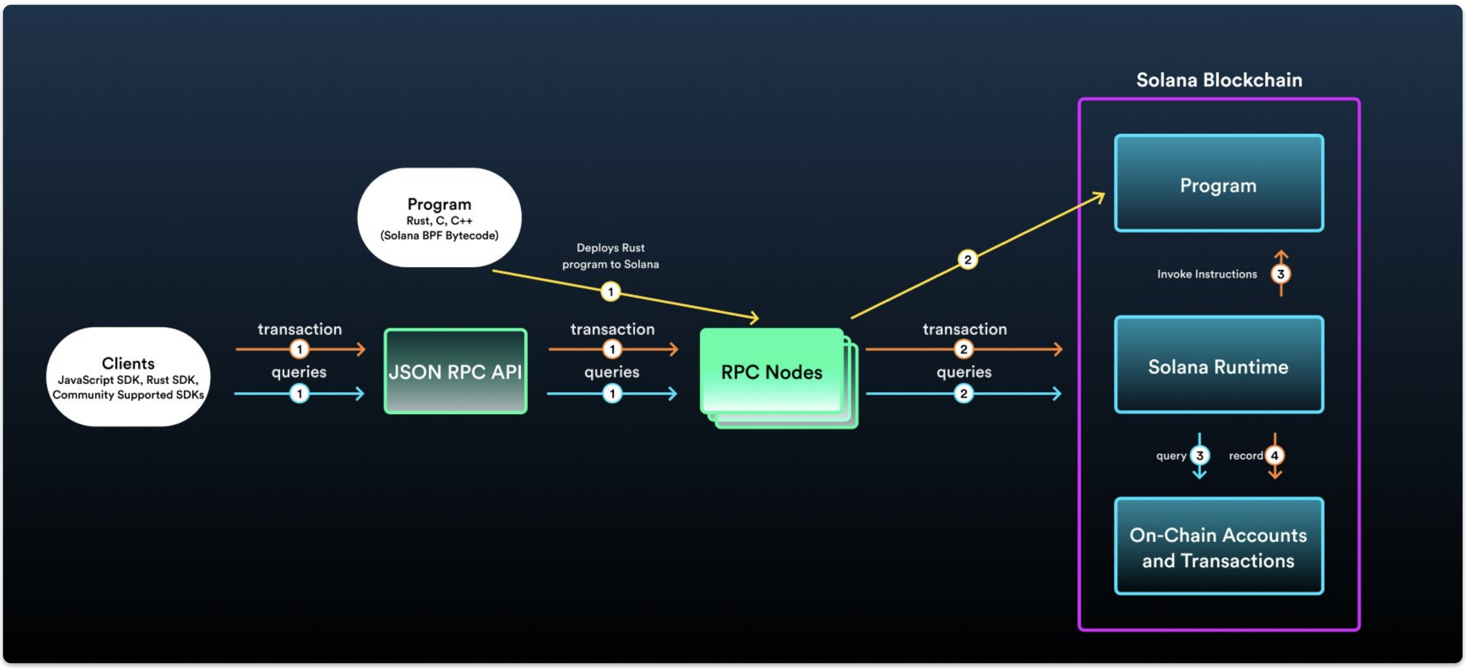


Cargo project architecture

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.

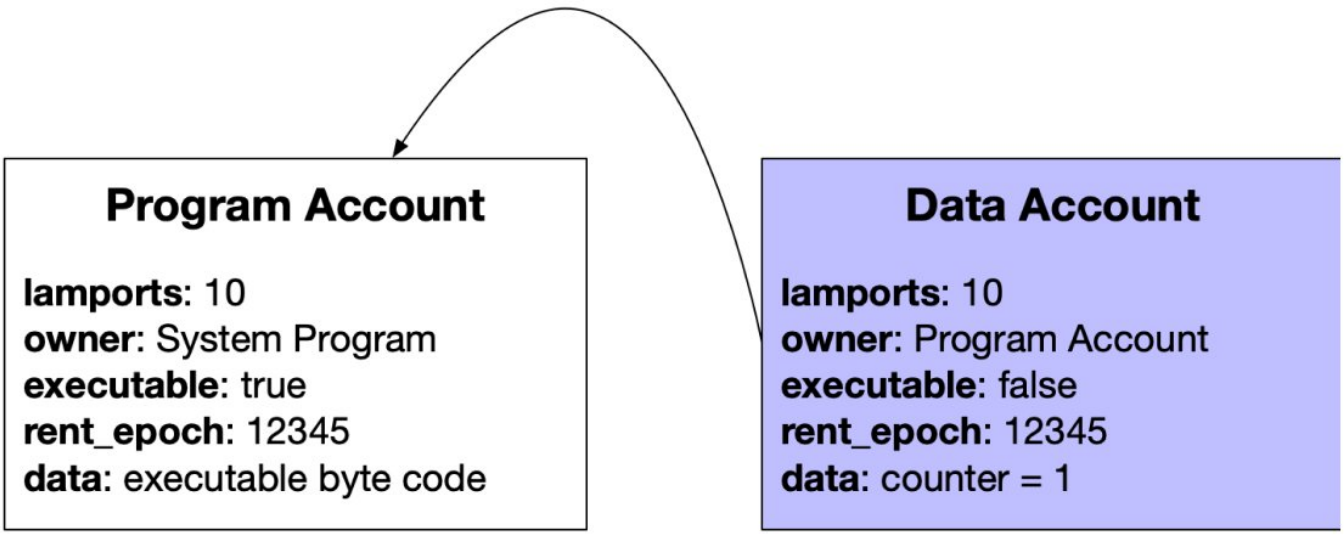
Interaction with RPC nodes is achieved using and SDK developed by solana labs.
This library (`@solana/web3.js`) abstracts away significant amount of boilerplate code meaning you can invoke functions rather than having to assemble each time JSON.

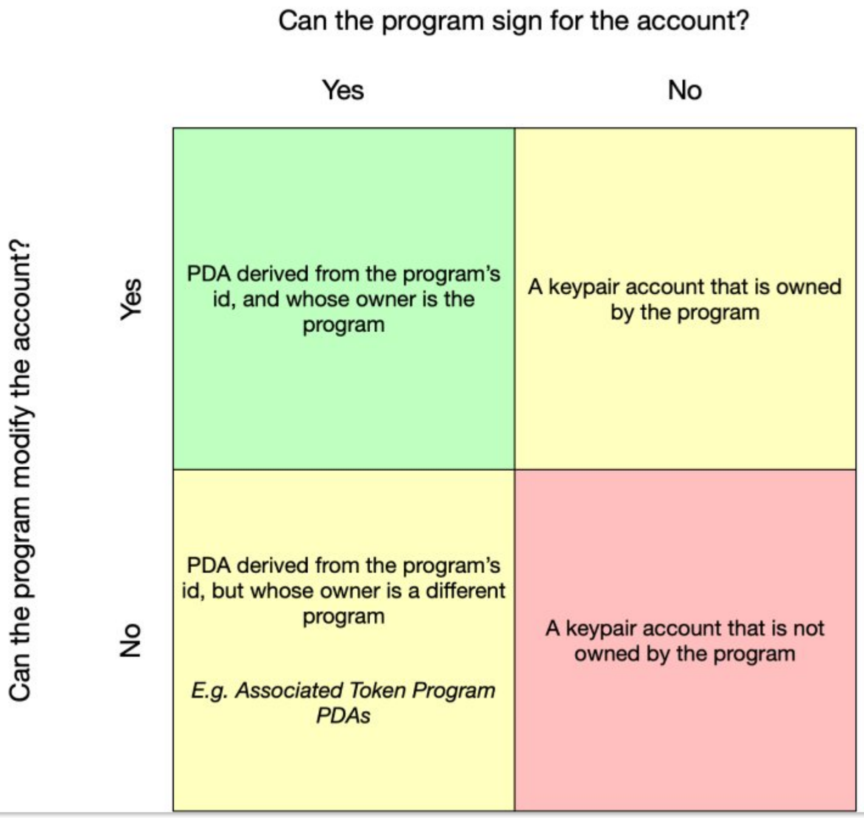
A list of available methods is here but these methods are not the same as the methods within a given program.
They are much broader and additional work including serialisation has to be done.
Thankfully there are libraries such as borsh or frameworks such as anchor to make it easier.

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: BPFLoaderUpgradeab1e11111111111111111111111
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 {
```

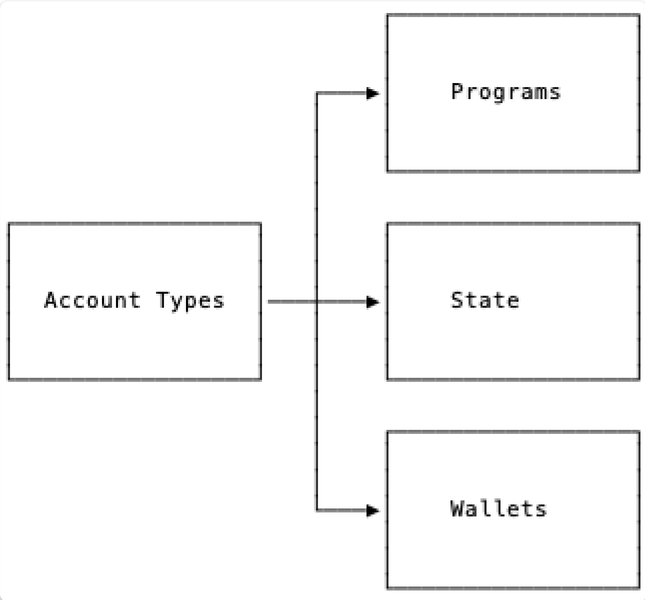
The program can return successfully, or with an error code. An error return causes the entire transaction to fail immediately.
On success you can not return any values, so in addition state has to be checked manually by the client.

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

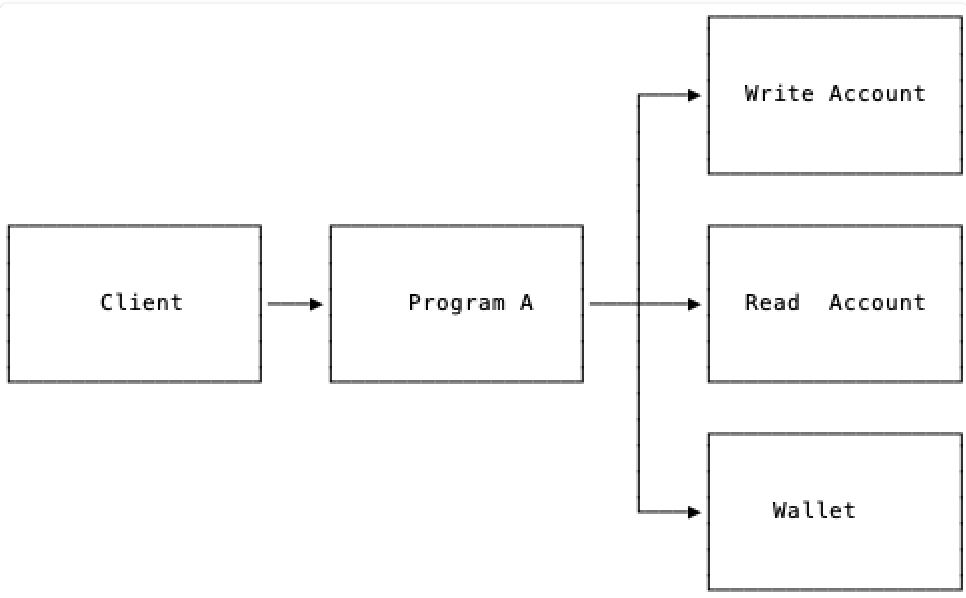
The accounts referenced by an instruction represent all the on chain accounts that this program will interact with and serve as both the inputs and outputs of a program. Account can be either a program containing logic, data account containing state or users wallet.



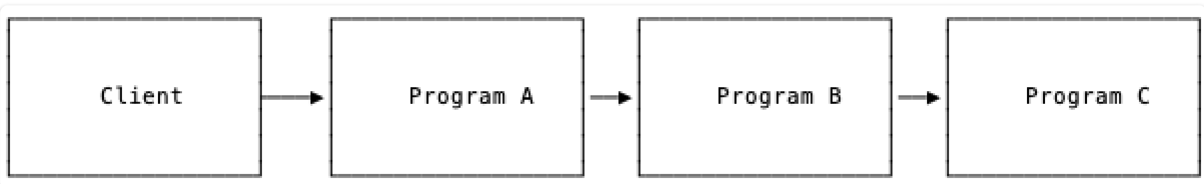
Account passed have to specify whether they will be read only or writeable.

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.

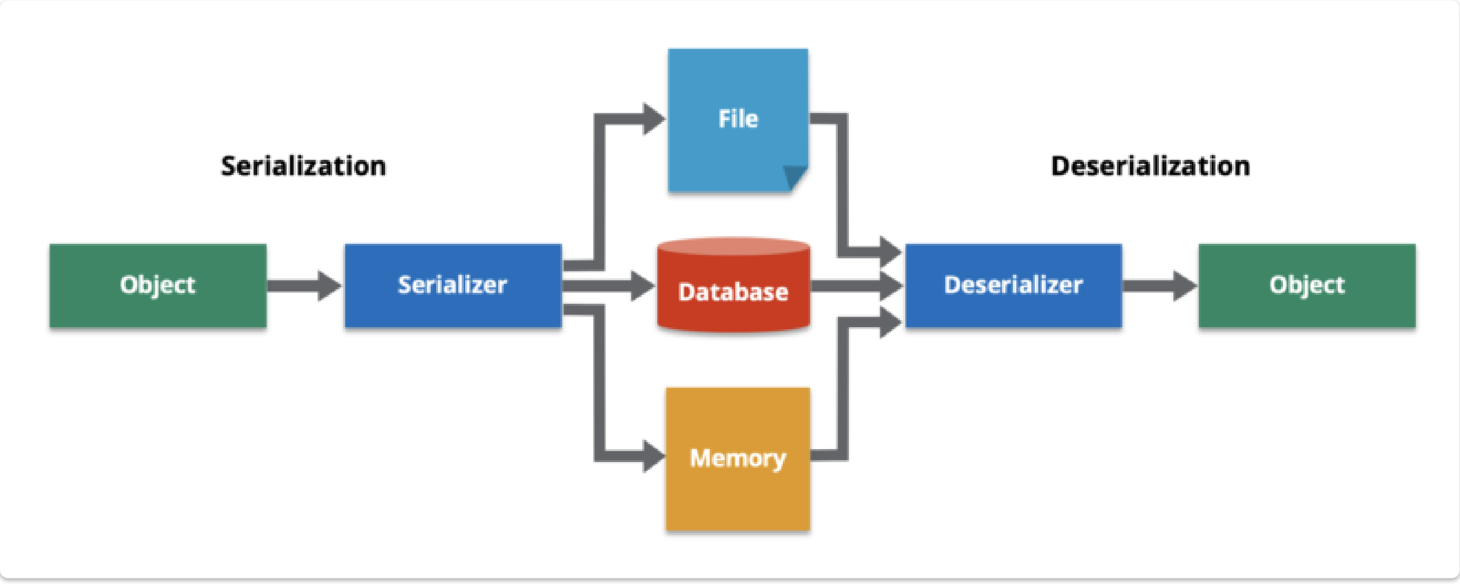


In future lessons we will look at Program Derived Addresses and how we use them when programs want to talk to each other.

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

Rust

Hashmaps

See Docs
and examples

Where vectors store values by an integer index, `HashMap`s store values by key. `HashMap` keys can be booleans, integers, strings, or any other type that implements the `Eq` and `Hash` traits.

Like vectors, `HashMap`s are growable, but HashMaps can also shrink themselves when they have excess space.
You can create a HashMap with a certain starting capacity using `HashMap::with_capacity(uint)`, or use `HashMap::new()` to get a HashMap with a default initial capacity (recommended).

An example inserting values

```
use std::collections::HashMap;
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

You can use an iterator to get values from the hashmap

```
let mut balances = HashMap::new();

balances.insert("132681", 12);
balances.insert("234987", 9);


for (address, balance) in balances.iter() {
        ...
}
```

USING THE GET METHOD

The get method on a hashmap returns an `Option<&V>` where V is the type of the value

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);
```

```
    let team_name = String::from("Blue");
    let score = scores.get(&team_name).copied().unwrap_or(0);
}
```

This program handles the `Option` by calling `copied` to get an `Option<i32>` rather than an `Option<&i32>`, then `unwrap_or` to set `score` to zero if `scores` doesn't have an entry for the key.

Traits

These bear some similarity to interfaces in other languages, they are a way to define the behaviour that a type has and can share with other types, where. behaviour is the methods we can call on that type.
Trait definitions are a way to group method signatures together to define a set of behaviours necessary for a particular purpose.

Defining a trait

```
pub trait Summary {
        fn summarize(&self) -> String;
}
```

Implementing a trait

```
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Default Implementations

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Using traits (polymorphism)

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Writing tests in Rust

See Docs
See examples

A test in Rust is a function that’s annotated with the `test` attribute

To change a function into a test function, add `#[test]` on the line before `fn`.
When you run your tests with the `cargo test` command, Rust builds a test runner binary that runs the annotated functions and reports on whether each test function passes or fails.

```
#[cfg(test)] mod tests {
#[test]
fn simple_example() {
        let result = 3 + 5;
        assert_eq!(result, 8);
        }
}
```

The `#[cfg(test)]` annotation on the tests module tells Rust to compile and run the test code only when you run `cargo test`, not when you run `cargo build`.