Lesson 5 - Solana Concepts / Rust errors

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.

Rust Errors etc.

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 ....");
```

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)?;
```