Lesson 13

This Week

Anchor
Solana Program Library
Metaplex

Today's Lesson

- Rust - lifetimes
- Solana - Program Composition reviewed
- Anchor high level description
- Anchor low level description
- Serialisation with and without anchor
    - Function calls
    - Accounts
    - Frequent implementations
        - Contexts

Rust - Lifetimes

See Docs

Every reference in Rust has a *lifetime*, which is the scope for which that reference is valid.
Most of the time, lifetimes are implicit and inferred

```
fn main() {
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
```

We would get an error message rom this code, since the reference created with
`r = &x;`
has gone out of scope when we try to print `r` on the last line.
The compiler uses the `borrow checker` to check that the lifetimes of references are valid.

This example will compile

```
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

since the data has a longer lifetime than the reference, we we can be sure that the reference will always refer to something valid.

A more complex example

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Here we have references as the function parameters since we don't want to take ownership. If we try to compile this, we get an error, because the compiler doesn't know whether the return is a reference to `x` or to `y` and the compiler cannot judge whether the references would always be valid.
To help the compiler, we need to be more explicit about the lifetimes involved.
To do this we use the lifetime annotation `'` followed by a parameter , for example
`'a`
This then follows the `&` in the reference to give for example
`&'a i32`
or
`&'a mut i32` for a mutable reference.

We can then use this annotation in our function signatures to specify the lifetimes of the parameters. For example

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

The function signature now tells Rust that for some lifetime `'a`, the function takes two parameters, both of which are string slices that live at least as long as lifetime `'a`.
The function signature also tells Rust that the string slice returned from the function will live at least as long as lifetime `'a`.

When we specify the lifetime parameters in this function signature, we're not changing the lifetimes of any values passed in or returned.
Rather, we're specifying that the borrow checker should reject any values that don't adhere to these constraints.
Note that the `longest` function doesn't need to know exactly how long `x` and `y` will live, only that some scope can be substituted for `'a` that will satisfy this signature.

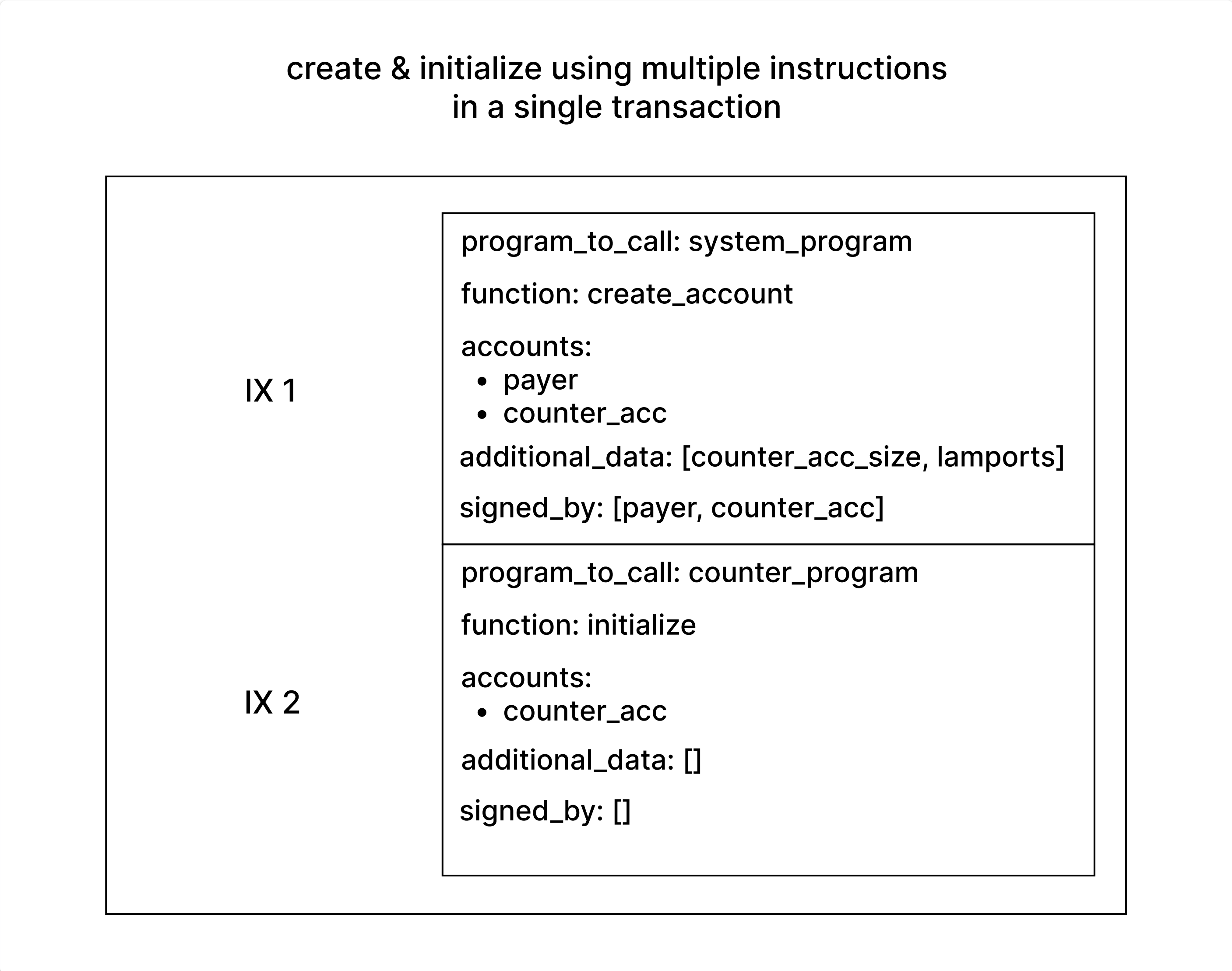
Program Composition Reviewed

Create & Initialize

Consider a counter program with two endpoints. One to initialize the counter and one to increment it. To create a new counter, we call the system program's `create_account` to create the account in memory and then the counter's `initialize` function.

Program Composition via multiple instructions in a transaction

The first way to create and initialize the counter is by using multiple instructions in a transaction. While a transaction can be used to execute a single call to a program, a single transaction can also include multiple calls to different programs.



If we went with this approach, our counter data structure would look like this:

```
pub struct Counter { pub count: u64, pub is_initialized: bool }
```

and our `initialize` function would look like this:

```
/// pseudo code fn initialize(accounts) { let counter = deserialize(accounts.counter); if counter.is_initialized { error("already initialized"); } counter.count = 0; counter.is_initialized = true; }
```

This approach could also be called the "implicit" approach. This is because the programs do not explicitly communicate with each other. They are glued together by the user on the client side.

This also means that the counter needs to have an `is_initialized` variable so `initialize` can only be called once per counter account.

Program Composition via Cross-Program Invocations

Cross-Program Invocations (CPIs) are the explicit tool to compose programs.
A CPI is a direct call from one program into another within the same instruction.

Using CPIs the create & initialize flow can be executed inside the `initialize` function of the counter:

`/// pseudo code fn initialize(accounts) { accounts.system_program.create_account(accounts.payer, accounts.counter); let counter = deserialize(accounts.counter); counter.count = 0; }`

In this example, no `is_initialized` is needed. This is because the CPI to the system program will fail if the counter exists already.

Anchor recommends CPIs to create and initialize accounts when possible
(Accounts that are created by CPI can only be created with a maximum size of `10` kibibytes.
This is large enough for most use cases though.).
This is because creating an account inside your own instruction means that you can be certain about its properties.
Any account that you don't create yourself is passed in by some other program or user that cannot be trusted.
This brings us to the next section.

Validating Inputs

On Solana it is crucial to validate program inputs. Clients pass accounts and program inputs to programs which means that malicious clients can pass malicious accounts and inputs. Programs need to be written in a way that handles those malicious inputs.

Consider the transfer function in the system program for example. It checks that `from` has signed the transaction.

/// simplified system program code

fn transfer(accounts, lamports) {
if !accounts.from.is_signer {
error();
}
accounts.from.lamports -= lamports;
accounts.to.lamports += lamports;
}

If it didn't do that, anyone could call the endpoint with your account and make the system program transfer the lamports from your account into theirs.

Consider the counter program from earlier. Now imagine that next to the counter struct, there's another struct that is a singleton which is used to count how many counters there are.

```
struct CounterCounter {
count: u64
}
```

Every time a new counter is created, the `count` variable of the counter counter should be incremented by one.

Consider the following `increment` instruction that increases the value of a counter account:

```
/// pseudo code
fn increment(accounts) {
let counter = deserialize(accounts.counter);
counter.count += 1;
}
```

This function is insecure.
It's not possible to pass in an account owned by a different program because the function writes to the account so the runtime would make the transaction fail.
But it is possible to pass in the counter counter singleton account because both the counter and the counter counter struct have the same structure (they're a rust struct with a single `u64` variable).
This would then increase the counter counter's count and it would no longer track how many counters there are.

The fix is:

```
/// pseudo code

let HARDCODED_COUNTER_COUNTER_ADDRESS = SOME_ADDRESS;

fn increment(accounts) {
if accounts.counter.key == HARDCODED_COUNTER_COUNTER_ADDRESS {
error("Wrong account type");
}
let counter = deserialize(accounts.counter);
counter.count += 1;
}
```

There are many types of attacks possible on Solana that all revolve around passing in one account where another was expected but it wasn't checked that the actual one is really the expected one.

Anchor High level

Anchor introduction

See Docs
See Anchor Book
Anchor is a framework for speeding up the development process of Solana smart contract design, testing and client interaction.

Anchor does this by:

- Abstracting serialisation of input parameters and account structure
- Abstracting RPC calls with function calls derived based on program IDL
- Abstracting additional checks as traits applied to contexts

Anchor Installation

See Docs

Via Cargo
`cargo install --git https://github.com/project-serum/anchor avm --locked --force`

then

```
avm install latest
avm use latest
```

Anchor configuration file

`Anchor.toml` file defines project parameters such as:

- Network for deployment/testing
- Programs managed by this `Anchor.toml`
- Custom scripts for testing, deployment or client interaction

This file gets autogenerated when initialising a new Anchor project and sits at the root of it.

Anchor Programs

An Anchor program consists of three parts.

1. The `program` module,
2. the Accounts structs which are marked with `#[derive(Accounts)]`, and
3. the `declare_id` macro.

The `program` module is where you write your business logic.
The Accounts structs is where you validate accounts.
The `declare_id` macro creates an `ID` field that stores the address of your program.
Anchor uses this hardcoded `ID` for security checks and it also allows other crates to access your program's address.

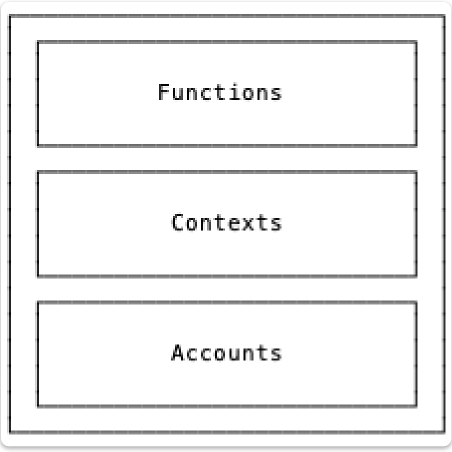
For example a boilerplate Anchor program would look like

```
// use this import to gain access to common anchor features
use anchor_lang::prelude::*;


// declare an id for your program
declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");


// write your business logic here
```

```
#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(_ctx: Context<Initialize>) -> Result<()> {
        Ok(())
    }
}


// validate incoming accounts here
#[derive(Accounts)]
pub struct Initialize {}
```

Anchor components

Every anchor program can be broken down into three distinct and functionally separate components: functions, contexts, and accounts.



Functions

These are single purposed logic blocks with deserialisation schema hidden behind the macros and upon successful invocation ending with changing of the state of at least one account.

Contexts

Contexts are lists of accounts that are to be passed to a given function, but also of actions that are to be executed in conjunction with these accounts and potentially to them.

Accounts

Accounts are blueprints for what a given on-chain account looks like once it is deserialised.

Anchor the Good and the Bad

Good:

- Much quicker to get to developing business logic instead of having fun with serialisation
- Extracts accounts provided within the context based on their name rather than the loading order and as such accidental usage of wrong accounts is less likely
- The same complexity can be achieved with a lot less boilerplate code
- No need for a module specifically for the deserialisation of instructions

Bad:

- Meaningless and confusing errors (also bare solana).
- Account deserialisation / creation fails before any custom logs can be logged to compare why it failed. As Solana runtime errors do not help much, it can be hard to pinpoint why an account initialisation macro failed.
- No granular control without running `anchor expand`
- Rust is in snake case but typescript IDL is in camel case, so you need to be careful not to call non existent methods
- Not very intuitive

Anchor in more detail

Serialisation abstraction for function calls

Without Anchor

Each bit has a purpose and is the responsibility of the developer to write out the boilerplate code that will modify the byte message train pattern to fit within what type of program functionality the client is invoking.

From example6-pda this is how at client side instruction data would be assembled for each of the function invocations.

Format for assembling instruction to call `create_pda` :

```
var instruction_data = Buffer.concat([
        Buffer.alloc(1, 0), // creating PDA
        Buffer.alloc(1, seed.length), // size of the seed (it varies)
        Buffer.from(seed), // seed buffer
        Buffer.alloc(1, bump), // bump integer
        Buffer.alloc(1, bytes), // acount size
]);
```

Output byte train for `instruction_data` to be sent to RPC node:

Function	Seed length	Seed[0]	...	Seed[n-1]	Bump	Account size

Format for assembling instruction to call `write_pda` :

```
var instruction_data = Buffer.concat([
        Buffer.alloc(1, 1), // function flag for writing PDA
        Buffer.alloc(1, word.length), // size of the word (it varies)
        Buffer.from(word), // bytes of user specified word
]);
```

Output byte train for `instruction_data` to be sent to RPC node:

Function	Word length	Word[0]	...	Word[n-1]

Any ordering and reordering of these bytes could be done to implement the same functionality. Using the first bit as a function flag makes sense from the point of view of a human, since accessing the last byte would be a very similiar action. The below example is just as valid provided deserialisation happens in the right order at the program side.

Function	Bump	Account size	Seed length	Seed[0]	...	Seed[n-1]

To note is the degree of flexibility that is required. There is no buffer to specify the size of the account beyond what a single byte of `u8` can hold. If the account required is to be above a certain size, this format would need to be modified to accommodate passing in byte arrays that can specify numbers above 255.

With Anchor

With anchor it is easy to call functions on chain 'directly' from the client.
No need for a specific decryption module (`instruction.rs`) nor for the client to manually write out instruction data.
All the serialisation still happens, but macros do it behind the curtain.

Example of a function declaration in Anchor program:

```
pub fn function(ctx: Context<Function>, <ARGUMENT>: <ARG_TYPE>) -> Result<()> {
```

Call from the client:

```
await program.methods
.function(<ARGUMENT>)
.accounts({[<ACCOUNTS>]})
.signers([[<SIGNERS>]])
.rpc()
```

Serialisation abstraction for accounts

Without Anchor

CLIENT SIDE

Describing an account template in typescript using Borsh:

```
...

class WordAccount {
	word = "";
	constructor(fields: { word: string } | undefined = undefined) {
		if (fields) {
			this.word = fields.word;
		}
	}
}

...

const WordSchema = new Map([
	[WordAccount, { kind: "struct", fields: [["word", "string"]] }],
]);

...
```

Calling RPC node with instruction including accounts in typescript:

Build total transaction instruction, including accounts and instruction data:

```
const instruction = new TransactionInstruction({
	keys: [
		{ pubkey: payer.publicKey, isSigner: true, isWritable: true },
		{ pubkey: theAccountToInit, isSigner: false, isWritable: true },
		{ pubkey: SystemProgram.programId, isSigner: false, isWritable:
false},
	],
	programId,
	data: instruction_data,
});
```

Submit transaction instruction to an RPC node:

await sendAndConfirmTransaction(

connection,
new Transaction().add(instruction),
[payer]
);

PROGRAM SIDE

Unpacking of the provided accounts:

As accounts are taken out one by one, the order in which they are fed by the client must match.

```
let mut accounts_iter = accounts.iter();
let acc1 = next_account_info(&mut accounts_iter)?;
let acc2 = next_account_info(&mut accounts_iter)?;
let acc3 = next_account_info(&mut accounts_iter)?;
let acc4 = next_account_info(&mut accounts_iter)?;
```

Declaration of a struct which describes account structure:

```
...

#[derive(BorshSerialize, BorshDeserialize, Debug, Clone)]
pub struct StringAccount {
        pub word: String,
}

...
```

This gets very tedious and error prone as variety of structs or growable collections are included within an account.

Deserialising an account using account template

As `StringAccount` has inherited from `BorshSerialize` in the `[derive(` macro it has access to its traits, like `try_from_slice` which attempts to deserialise a given account based on the layout of `StringAccount`.

```
let mut string_acc = StringAccount::try_from_slice(&h.data.borrow())?;
```

Serialisation of the new state using Borsh's `serialize` trait:

Opposite action to the one above, calling `serialize` trait on an instance of a local `StringAccount` handle `string_acc`.

```
string_acc.serialize(&mut &mut hello_account.data.borrow_mut()[..])?;
```

With Anchor

For source code see `examples_anchor/programs/example1-lottery` in the repo.

Some libraries such as borsh make this easier for accounts and are used throughout the baremetal examples. Anchor goes a step further with the abstraction.

In the program accounts are described as such:

```
#[account]
pub struct Lottery {
        pub authority: Pubkey,
        pub oracle: Pubkey,
        pub winner: Pubkey,
        pub winner_index: u32,
        pub count: u32,
        pub ticket_price: u64,
}
```

The client knows the structure function and account structure from interface description language (IDL) made at the compilation time, this is similar to the ABI in solidity.

Client side code for fetching the right account (IDL)

Setup

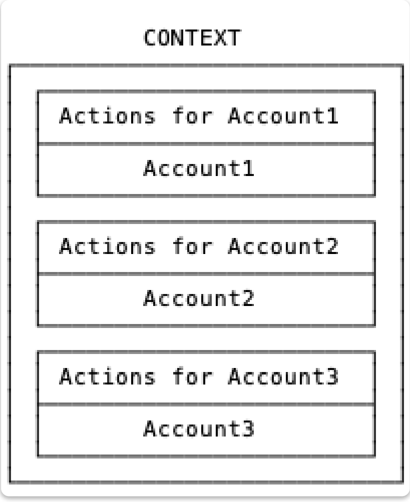
```
const provider = anchor.AnchorProvider.local();
anchor.setProvider(anchor.AnchorProvider.local());
const program = anchor.workspace.<PROGRAM_NAME>;
```

Now reading account state is much easier and a necessary step to ensure the correct functionality. This snippet will retrieve the value of field `winnerIndex` of an account imported under the name `lottery` at the address of `lottery.publicKey`:

```
await program.account.lottery.fetch(lottery.publicKey).winnerIndex;
```

Abstraction of frequent implementations

Macros are also used to hide tedious interactions relating to creating, validating and writing to an on-chain account. These are applied inside the context and anything from that context can be accessed inside a given function.



As a given context is per individual function call, different function calls can have different ways to interact with the same account.
Any accounts not explicitly defined and processed within the context, can be attached as a vector of accounts at client by adding the `.remainingAccounts()` invocation.

Doing that you need to state whether these accounts are to be mutable and whether they will have a signature attached as there is no context from which that information could be obtained.

```
await program.methods
.METHOD_NAME()
.accounts({
        ACC_1: ACC_1_TEMPLATE, // Accounts
        ACC_2: ACC_2_TEMPLATE, // defined in
        ACC_3: ACC_3_TEMPLATE, // context
})
.remainingAccounts([
{
        pubkey: ACC_4, isWritable: false, isSigner: false,
        pubkey: ACC_5, isWritable: false, isSigner: false,
        pubkey: ACC_6, isWritable: false, isSigner: false,
},
]).signers([])
.rpc();
```

Context example: Create

Function

```
pub fn initialise_lottery(ctx: Context<Create>, ticket_price: u64) -> Result<()> {
```

Context

```
#[derive(Accounts)]
pub struct Create<'info> {
```

```
        #[account(init, payer = admin, space = 180)]
        pub lottery: Account<'info, Lottery>,
        #[account(mut)]
        pub admin: Signer<'info>,
        /// CHECK:
        pub oracle: UncheckedAccount<'info>,
        pub system_program: Program<'info, System>,
}
```

This context describes following actions:

- Account `lottery` (not-PDA) of the `Lottery` blueprint is to be initialised.
- The payer will be the account `admin` which provides signature to authorise the payment and is marked as mutable.
- Space set aside for that account is to be 180 bytes.
- Address for an oracle is provided, it does not have any macros applied and is non-mutable by default.
- `system_program` is necessary when requesting a creation of a new account from program

---

Context example: Submit

Function

```
pub fn add_submission(ctx: Context<Submit>) -> Result<()> {
```

Context

```
#[derive(Accounts)]
pub struct Submit<'info> {
        #[account(init,
        seeds = [
                &lottery.count.to_be_bytes(),
                lottery.key().as_ref()
        ],
        bump,
        payer = player,
        space=80)
        ]
        pub submission: Account<'info, Submission>,
        #[account(mut)]
        pub player: Signer<'info>,
        pub system_program: Program<'info, System>,
        #[account(mut)]
        pub lottery: Account<'info, Lottery>,
}
```

This context describes following actions:

- Account `submission` (PDA) of the `Submission` blueprint is to be initialised.
- PDA derivation `seeds` format will be based on lottery counter and lottery address
- The payer will be the account `player` which provides signature to authorise payment and is marked as mutable.
- Space set aside for that account is to be 80 bytes.
- `system_program` is necessary when requesting a creation of a new account from program
- `lottery` account is marked as mutable as:
    - value of the counter needs to be read to be used in the submission PDA derivation
    - value of the counter has to be incremented

Context example: Winner

Function

```
pub fn pick_winner(ctx: Context<Winner>, winner: u32) -> Result<()> {
```

Context

#[derive(Accounts)]
pub struct Winner<'info> {
#[account(mut, constraint = lottery.oracle == *oracle.key)]
pub lottery: Account<'info, Lottery>,
pub oracle: Signer<'info>,
}

The above context describes following actions:

- Account `lottery` (PDA) of the `Lottery` blueprint:
  - has `mut` applied as it is to be written to
  - will only procceed if `constraint` is satisified, which is that the caller is the ceritifed oracle
- Account passed in as `oracle` needs to include signature

Additional account macros

Full list of possible macros is too large to include here, but can be found here.

Attribute	Description

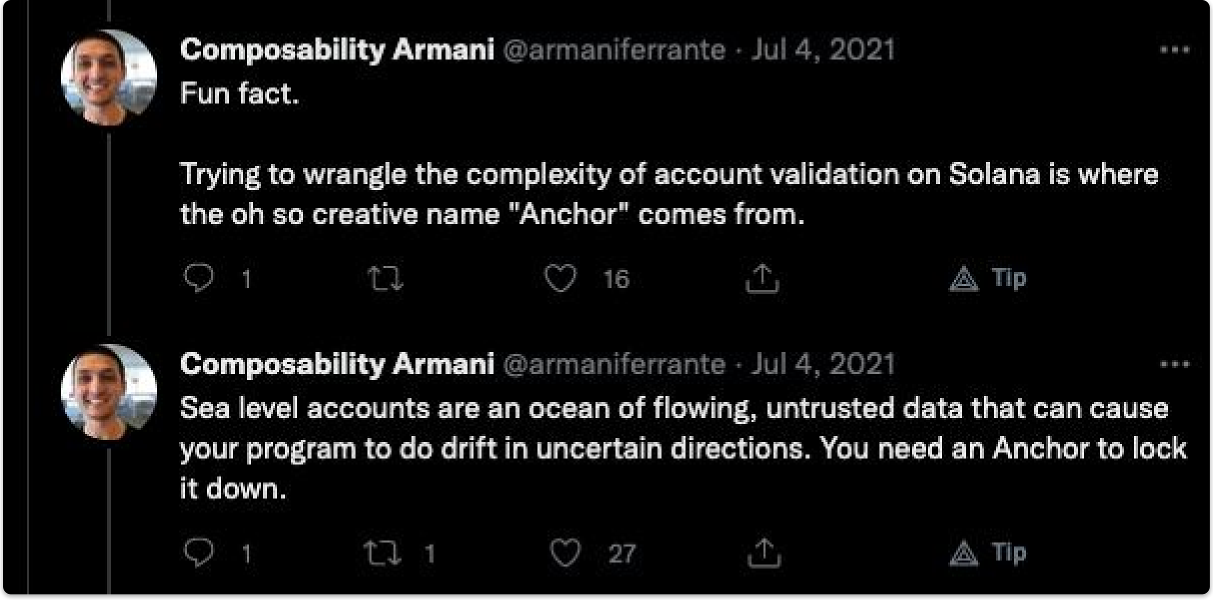
`#[account(signer)]`
 	
`#[account(signer @ <custom_error>)]`
	Checks the given account signed the transaction.
	Custom errors are supported via `@`.
	Consider using the `Signer` type if you would only have this constraint on the account.
	
	Example:
	
	`#[account(signer)]`
	`pub authority: AccountInfo<'info>,`
	`#[account(signer @ MyError::MyErrorCode)]`
	`pub payer: AccountInfo<'info>`

`#[account(mut)]`
 	
`#[account(mut @ <custom_error>)]`
	Checks the given account is mutable.
	Makes anchor persist any state changes.
	Custom errors are supported via `@`.
	
	Example:
	
	`#[account(mut)]`
	`pub data_account: Account<'info, MyData>,`
	`#[account(mut @ MyError::MyErrorCode)]`
	`pub data_account_two: Account<'info, MyData>`

`#[account(init, payer = <target_account>, space = <num_bytes>)]`	Creates the account via a CPI to the system program and initializes it (sets its account discriminator). Marks the account as mutable and is mutually exclusive with `mut`. Makes the account rent exempt unless skipped with `rent_exempt = skip`. Use `#[account(zero)]` for accounts larger than 10 Kibibyte. `init` has to be used with additional constraints: • Requires the `payer` constraint to also be on the account. The `payer` account pays for the account creation. • Requires the system program to exist on the struct and be called `system_program`. • Requires that the `space` constraint is specified. When using the `space` constraint, one must remember to add 8 to it which is the size of the account discriminator. This only has to be done for accounts owned by anchor programs. The given space number is the size of the account in bytes, so accounts that hold a variable number of items such as a `Vec` should allocate sufficient space for all items that may be added to the data structure because account size is fixed. Check out the space reference and the borsh library (which anchor uses under the hood for serialization) specification to learn how much space different data structures require. Example:

Anchor name origin

Origins of the name anchor according to the creator.


Composability Armani @armaniferrante · Jul 4, 2021
Fun fact.
Trying to wrangle the complexity of account validation on Solana is where the oh so creative name "Anchor" comes from.
1
16
Tip
Composability Armani @armaniferrante · Jul 4, 2021
Sea level accounts are an ocean of flowing, untrusted data that can cause your program to do drift in uncertain directions. You need an Anchor to lock it down.
1
1
27
Tip