Lesson 14

Today's Lesson - Anchor Continued / SPL

- Anchor review
- Contexts
- Testing
  - Addresses
  - Scripts
  - Logs
  - State checks
  - Legacy Tools
- SPL introduction

Anchor review

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.

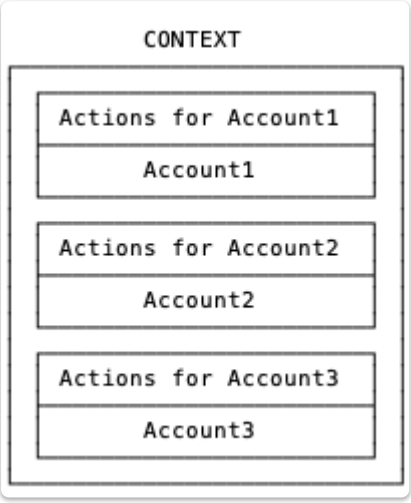
The `program` module is where you write your business logic.
The Accounts structs is where you validate accounts.
The `declare_id` macro creates an `ID` field that stores the address of your program.
Anchor uses this hardcoded `ID` for security checks and it also allows other crates to access your program's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

Contexts

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

---

Constraints

Add constraints with

```
#[account(<constraints>)]
pub account: AccountType
```

For example

```
#[derive(Accounts)]
pub struct SetData<'info> {
    #[account(mut)]
    pub my_account: Account<'info, MyAccount>,
    #[account(
        constraint = my_account.mint == token_account.mint,
        has_one = owner
    )]
    pub token_account: Account<'info, TokenAccount>,
    pub owner: Signer<'info>
}
```

Two of the Anchor account types, AccountInfo and UncheckedAccount do not implement any checks on the account being passed.
Anchor implements safety checks that encourage additional documentation describing why additional checks are not necessary.

These can produce an error which is fixed with

```
#[derive(Accounts)]
pub struct Initialize<'info> {
    /// CHECK: This is not dangerous because we don't read or write from this
account
    pub potentially_dangerous: UncheckedAccount<'info>
}
```

Custom Errors

You can add errors that are unique to your program by using the `error_code` attribute.

Simply add it to an enum with a name of your choice. You can then use the variants of the enum as errors in your program. Additionally, you can add a message attribute to the individual variants. Clients will then display this error message if the error occurs.

To actually throw an error use the `err!` or the `error!` macro. These add file and line information to the error that is then logged by anchor.

```
#[program]
mod hello_anchor {
    use super::*;
    pub fn set_data(ctx: Context<SetData>, data: MyAccount) -> Result<()> {
        if data.data >= 100 {
            return err!(MyError::DataTooLarge);
        }
        ctx.accounts.my_account.set_inner(data);
        Ok(())
    }
}


#[error_code]
pub enum MyError {
    #[msg("MyAccount may only hold data below 100")]
    DataTooLarge
}
```

Require !

You can use the `require` macro to simplify writing errors. The code above can be simplified to this

```
#[program]
mod hello_anchor {
    use super::*;
    pub fn set_data(ctx: Context<SetData>, data: MyAccount) -> Result<()> {
        require!(data.data < 100, MyError::DataTooLarge);
        ctx.accounts.my_account.set_inner(data);
        Ok(())
    }
}


#[error_code]
pub enum MyError {
    #[msg("MyAccount may only hold data below 100")]
    DataTooLarge
}
```

Testing with anchor

Dev loop

Anchor includes several ways you can test your code, it can redeploy upon each test run or call pre deployed program.
The network for testing can also be specified and it will be either localnet or devnet.
The developer has a finer control over localnet as well as faster response, but if the program being tested is complex and requires calls to many other programs it makes sense to test on devnet rather than having to redeploy all these necessary contracts.

Just like with baremetal development, the process is an iteration of following steps.

- build program(s) achieved with `anchor build`
- deploy program(s) with `anchor deploy`
- run test(s) with `anchor run test`

Modifying the tests, requires only rerunning of the `anchor run test`.
Modifying the program, requires rerunning of every command.

To compile, deploy and tests in one command `anchor test` can be run, though it will take longer due to deployment time and it will use up more lamports.

Hardcoding address

The first time a Solana program is built, the compiler will generate a byte array which will be used as a deployment key and from it will be derived the public key of that program account.
It will be right next to the `.so` binary with it's name derived from program's name in the `Cargo.toml`.

Program `name` in `Cargo.toml` decides file names:

```
...

[package]
name = "name"

...
```

Above `Cargo.toml` will produce following files:

- `name.so` program binary
- `name-keypair.json` program deployment seed

Anchor requires hardcoding program account address into the actual program (so it has some state) and this can't be known until the first deployment.
This makes it easier to reference packages and includes additional security checks.

What this means is that `anchor build` and `anchor deploy` will have to be run twice.

After first deployment we will need to get the `Program Id` (program's address) from the terminal output:

```
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
msg 5r9zXFR1pnNhwSxFQAvRhzapYmxFMR8B9K8Zcqx9pSr4
Program Id: 6zMoFAAdk3EAE4akCNXSnoGqra5ZU8ZhYaSucY31WbsW
```

This public key needs to be copied and pasted into:

1. `Anchor.toml` project configuration

```
[programs.localnet]
name = "6zMoFAAdk3EAE4akCNXSnoGqra5ZU8ZhYaSucY31WbsW"
```

2. Program being developed into the `declare_id!` macro

```
use anchor_lang::prelude::*;
use anchor_lang::solana_program::hash::hash;
use anchor_lang::solana_program::hash::Hash;

▶ Run Test | Debug
declare_id!("6zMoFAAdk3EAE4akCNXSnoGqra5ZU8ZhYaSucY31WbsW");

#[program]
mod name {
    use super::*;
```

After this is done the program needs to be recompiled and redeployed.
If `name-keypair.json` file is deleted upon new compilation another keypair will be generated and same process will need to be redone.

Anchor scripts

A testing script can be modified to run a specific test only if a finer testing granularity is needed. Scripts can be used not only for testing but for deployment or to start client.

Example of `Anchor.toml` scripts:

```
[scripts]
test = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/*.ts"
test1 = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/test_ex
test2 = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/test_ex
test3 = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/test_ex
```

Script `test` will run all `.ts` files in the test directory, whilst other ones will only test a specific file. To run the script using anchor.tml

`anchor run <SCRIPTR_NAME>`

Logs

Program variables can still be logged with the `msg!` macro to a terminal window running `solana logs`.

Client side state checks

Account state needs to be checked at the end of a test to ensure correct state transition.

The following syntax will load local `account` instance based on its IDL:

```
let account = await program.account.<ACC_NAME>.fetch(<ACC_KEY>)
```

Fields of that account can be accessed like this:

```
let winner = account.winner
let loser = account.loser
let authority = account.auth
```

Validity can be checked using `chai`:

```
expect(winner).to.not.equal(loser);
```

Usage of legacy Solana typescript tools

Anchor library inherits from `"@solana/web3.js"` so all the standard functions can be accessed such as airdrops or balance checks.
RPC calls can be performed without utilising underlying Anchor serialisation tools and somethings at client side may be easier to achieve without Anchor.

Import in non-Anchor test:

```
import {
    Connection,
    Keypair,
    PublicKey,
    TransactionInstruction,
    sendAndConfirmTransaction,
    Transaction,
} from "@solana/web3.js";
```

Import in Anchor test:

```
import * as anchor from "@project-serum/anchor";
const { SystemProgram, PublicKey } = anchor.web3;
```

SPL

Solana Program Library covers

- tokens
- governance
- name service
- token swaps
- lending

The Solana token program is heavily used, its program id is
: TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA

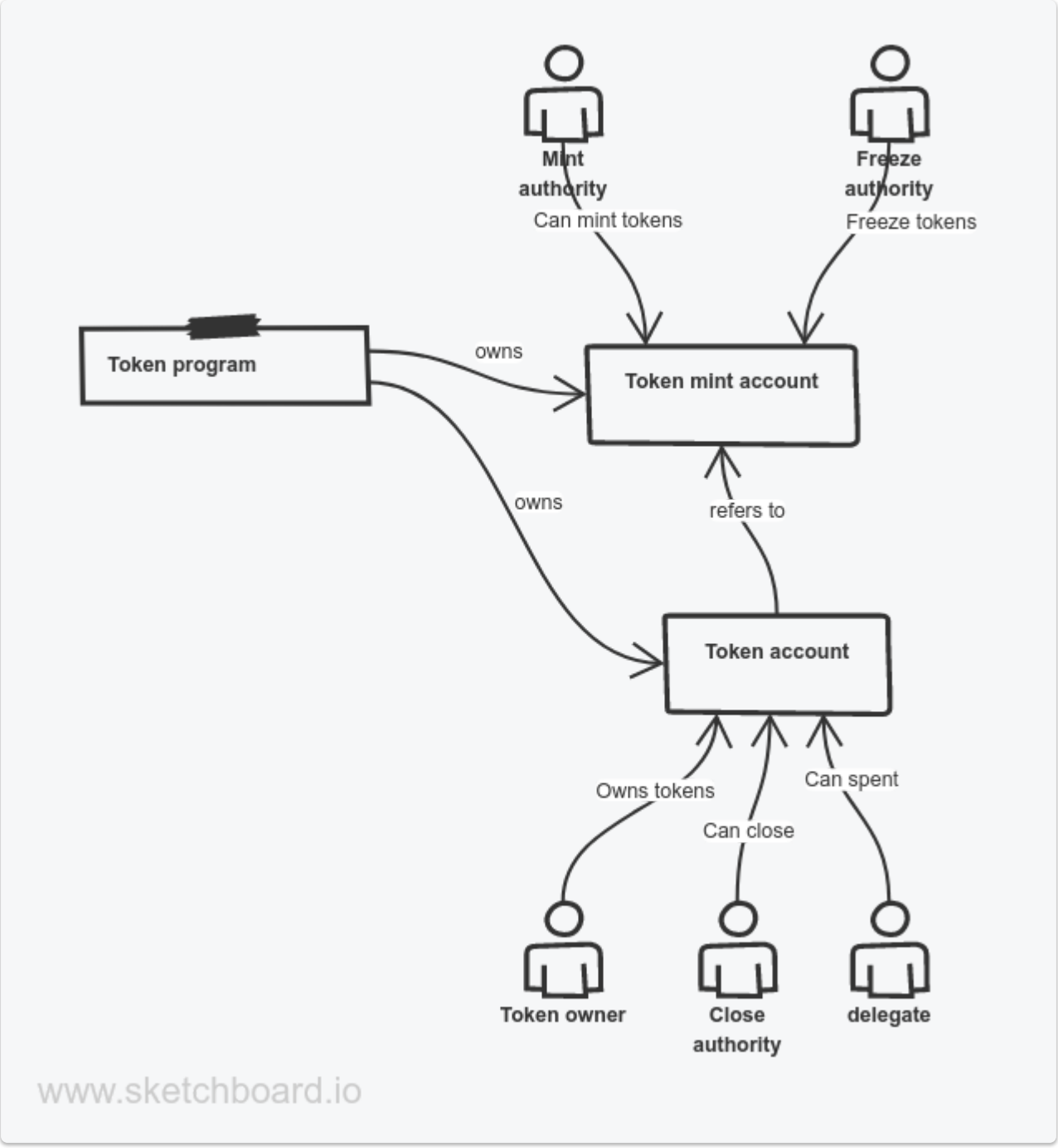
Documentation can be found at docs

Token Program

Creating a new token type

A new token type can be created by initialising a new Mint with the `InitializeMint` instruction. The Mint is used to create or "mint" new tokens, and these tokens are stored in Accounts. A Mint is associated with each Account, which means that the total supply of a particular token type is equal to the balances of all the associated Accounts.

The process of creating a new token type



We first call `InitializeMint`
This takes some parameters that are stored in a struct

```
pub struct Mint {
/// Optional authority used to mint new tokens. The mint authority may only be prov
pub mint_authority: COption<Pubkey>,

/// Total supply of tokens.
pub supply: u64,

/// Number of base 10 digits to the right of the decimal place.
pub decimals: u8,

/// Is `true` if this structure has been initialized
pub is_initialized: bool,

/// Optional authority to freeze token accounts.
pub freeze_authority: COption<Pubkey>,
```

```
}
```

Creating accounts

Accounts hold token balances and are created using the `InitializeAccount` instruction. Each Account has an owner who must be present as a signer in some instructions.

An Account's owner may transfer ownership of an account to another using the `SetAuthority` instruction.

It's important to note that the `InitializeAccount` instruction does not require the Solana account being initialised also be a signer. The `InitializeAccount` instruction should be atomically processed with the system instruction that creates the Solana account by including both instructions in the same transaction.

```
pub struct Account {
/// The mint associated with this account
pub mint: Pubkey,
/// The owner of this account.
pub owner: Pubkey,
/// The amount of tokens this account holds.
pub amount: u64,
/// If `delegate` is `Some` then `delegated_amount` represents
/// the amount authorized by the delegate
pub delegate: COption<Pubkey>,
/// The account's state
pub state: AccountState,
/// If is_native.is_some, this is a native token, and the value logs the rent-exemp
/// Account is required to be rent-exempt, so the value is used by the Processor to
/// wrapped SOL accounts do not drop below this threshold.
pub is_native: COption<u64>,
/// The amount delegated
pub delegated_amount: u64,
/// Optional authority to close the account.
pub close_authority: COption<Pubkey>,

}
```

Next the `MintTo` instruction is called, taking

- Public key of the mint
- Address of the token account to mint to
- The mint authority
- Amount to mint
- Signing accounts if `authority` is a multisig
- SPL Token program account

This will mint tokens to the destination account

TRANSFER

To transfer tokens we invoke the function `process_transfer` this transfers a certain amount of token from a source account to a destination account:
We pass in the source and destination accounts and the amount.
The program will check that

1. Neither source account nor destination account is frozen
2. The source account's mint and destination account's mint are the same
3. The transferred `amount` is no more than source account's token amount

The owner of the source Account must be present as a signer in the `Transfer` instruction when the source and destination accounts are different.

Note the source and destination can be the same, if so the `Transfer` will *always* succeed.
Therefore, a successful `Transfer` does not necessarily imply that the involved Accounts were valid SPL Token accounts, that any tokens were moved, or that the source Account was present as a signer.
It is recommended to check that the source and destination are different before calling the transfer function.

BURN

Burn is the opposite of Mint and removes tokens, from the supply and the given account.

APPROVE

This allows transfer of a certain amount by a delegate.

- Only one delegate is possible per account / token.
- A new approval will overrride the previous one.

REVOKE

Removes the approval

FREEZE / THAW ACCOUNT

This will freeze / unfreeze the account preventing / allowing transfers / mints to it.

Associated Token Program

From Solana Docs
A user may own arbitrarily many token accounts belonging to the same mint which makes it difficult for other users to know which account they should send tokens to and introduces friction into many other aspects of token management.
The associated token program introduces a way to deterministically derive a token account key from a user's main System account address and a token mint address, allowing the user to create a main token account for each token they own.

We call these accounts *Associated Token Accounts*.

In addition, it allows a user to send tokens to another user even if the beneficiary does not yet have a token account for that mint. Unlike a system transfer, for a token transfer to succeed the recipient must have a token account with the compatible mint already, and somebody needs to fund that token account. If the recipient must fund it first, it makes things like airdrop campaigns difficult and just generally increases the friction of token transfers.
The associated token program allows the sender to create the associated token account for the receiver, so the token transfer just works.

The associated token account for a given wallet address is simply a program-derived account from the wallet address itself and the token mint.
This gives us a way to deterministically find an account address based on the wallet and the mint account.

FINDING THE ASSOCIATED TOKEN ADDRESSES

The get_associated_token_address Rust function may be used by clients to derive the wallet's associated token address.

The associated account address can be derived in TypeScript with:

```
import { PublicKey } from '@solana/web3.js';
import { TOKEN_PROGRAM_ID } from '@solana/spl-token';

const SPL_ASSOCIATED_TOKEN_ACCOUNT_PROGRAM_ID: PublicKey = new PublicKey(
  'ATokenGPvbdGVxr1b2hvZbsiqW5xWH25efTNsLJA8knL',
);

async function findAssociatedTokenAddress(
    walletAddress: PublicKey,
    tokenMintAddress: PublicKey
): Promise<PublicKey> {
    return (await PublicKey.findProgramAddress(
        [
            walletAddress.toBuffer(),
            TOKEN_PROGRAM_ID.toBuffer(),
            tokenMintAddress.toBuffer(),
        ],
        SPL_ASSOCIATED_TOKEN_ACCOUNT_PROGRAM_ID
    ))[0];
}
```

Creating the Associated Token Account

If the associated token account for a given wallet address does not yet exist, it may be created by *anybody* by issuing a transaction containing the instruction returned by create_associated_token_account.

Regardless of creator the new associated token account will be fully owned by the wallet, as if the wallet itself had created it.

This article looks at tokens and accounts in depth, and has some good diagrams to show the relationship between the different accounts.