Anchor Introduction

Anchor is a framework for building Solana programs.

The Anchor framework organizes a program into distinct sections that separates the instruction logic from account validation and security checks. Anchor speeds up the process of building Solana programs by abstracting away a significant amount of boilerplate code.

For this lesson, you will not need to copy/paste any code. We'll walkthrough how to initialize an Anchor project using the command line. One of the benefits of using Anchor is that it will initialize an entire project with some starter code from the CLI!

Why Anchor

With Anchor you can build programs quickly because it writes various boilerplate for you such as (de)serialization of accounts and instruction data.

You can build secure programs more easily because Anchor handles certain security checks for you. On top of that, it allows you to succinctly define additional checks and keep them separate from your business logic.

That means no more worrying about manually converting instruction data from raw byte buffers to a data struct so that you can understand it. Anchor handles this automatically.

You also do not have to spend time writing code to just validate the accounts passed in instead of working on the business logic of your program. Anchor handles some of this automatically, but also makes adding customized account validations trivial.

Both of these aspects mean that instead of working on the tedious parts of raw Solana programs, you can spend more time working on what matters most, your product.

Components of Anchor

Anchor programs have three main components:

1. `declare_id!` macro
2. the `program` module
3. the `#[derive(Accounts)]` struct

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.

```
// 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 {}
```

Introducing Macros

Anchor is able to make Solana development much easier and more efficient by making use of Macros, which are a feature that's unique to the Rust language.

Macros essentially write more code behind the scenes than what you have written yourself manually. This reduces the amount of code you have to write and also speeds up development.

The syntax for utilizing a macro will always be `<macro name>!(<input data>)`.

To learn more about macros and other Rust primitives. The `declare_id!` section of an Anchor program makes use of a macro.

You can actually see the code that macros are writing behind the scenes by typing `cargo expand` in the terminal inside a Rust program. Anchor abstracts away some boilerplate code of Solana development with macros.

Anchor has a default public key that it will place in the `declare_id!` macro once an Anchor project is initialized. Once you deploy an Anchor program with the CLI, a public key for the program will be generated and printed to the command line.

This is the actual public key of your program, so you'll have to be sure to go back and update the `declare_id!` macro once you've deployed it. We'll cover this later once we actually deploy our own program.

```
// default public key when Anchor project is initialized
declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");
```

Modules Are Where the Magic Happen

The `program` module is where the business logic of your program lives. These are the nuts and bolts of the contract, the whole reason for even writing a smart contract.

Modules are a way of organizing and grouping code in Rust, similar to classes in Python. In Anchor, we define our `program` module and then put any instructions that we want to be callable from outside the contract here.

These are the functions that transactions submitted to the blockchain will be targeting once the program is deployed. You can also think of these functions as the endpoints to our API.

```
// simple example from Anchor github repo
#[program]
mod basic_2 {
    use super::*;

    pub fn create(ctx: Context<Create>, authority: Pubkey) -> Result<()> {
        let counter = &mut ctx.accounts.counter;
        counter.authority = authority;
        counter.count = 0;
        Ok(())
    }

    pub fn increment(ctx: Context<Increment>) -> Result<()> {
        let counter = &mut ctx.accounts.counter;
        counter.count += 1;
        Ok(())
    }
}
```

Accounts Struct

The Accounts struct is where you define which accounts your instruction expects and which constraints these accounts should adhere to. Remember that every instruction requires all the accounts involved to be sent.

It’s the same on the program side.

When writing a Solana program, each instruction must declare ahead of time the accounts it expects to receive. This makes it very easy to determine when incorrect accounts have been sent.

We won’t dive too deep into the Accounts struct in this lesson, we will cover this more in the next one.

Let's reflect briefly on what we've learned thus far. Then we can get coding.

Hello World

To initialize an Anchor project, you’ll need to move to the directory where you want to store this project on your computer. Anchor comes with its own CLI that’s similar to the Solana CLI. To create a new Anchor project, simple run `anchor init <new-workspace-name>` in your terminal.

```
anchor init hello-world
```

Anchor creates a new directory under the project name you provided, open up your newly created directory in your code editor of choice. You may notice that there are a lot of different files and sub-directories listed here - this is the default structure of an Anchor project, it's often referred to as a Workspace.

The following are some of the important files and folders of the workspace:

- The `.anchor` folder: It includes the most recent program logs and a local ledger that is used for testing
- The `app` folder: An empty folder that you can use to hold your frontend if you use a mono-repo
- The `programs` folder: This folder contains your programs. It can contain multiple but initially only contains a program with the same name as `<new-workspace-name>`. This program already contains a `lib.rs` file with some sample code.
- The `tests` folder: The folder that contains your E2E tests. It will already include a file that tests the sample code in the `programs/<new-workspace-name>`.
- The `migrations` folder: In this folder you can save your deploy and migration scripts for your programs.
- The `Anchor.toml` file: This file configures workspace wide settings for your programs.

Inside the `programs` directory, there should be a lib.rs file already created. This single file will serve as our Solana program for now. You'll notice there is already some code in the file and that it follows the exact same layout that we learned about in the beginning of this lesson.

The sample code doesn't really do anything right now, but it's helpful to not have to write your programs from scratch sometimes. To turn this into a hello world program, all we need to do is add a log in the `initalize` function to say hi. The program code should look like this.

```
use anchor_lang::prelude::*;

declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");

#[program]
pub mod hello_world {
    use super::*;

    pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
        msg!("Hello Metacrafter!");
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize {}
```

All this program will do is log "Hello Metacrafter!" when an instruction is sent to it.

Another feature of Anchor is it makes testing your programs very easy! If you open up the file in your tests directory of the workspace, you'll see there is already some Typescript code there that tests the default

program! Your code editor may complain about the path on this import missing, but that's okay for now. This line is just importing the IDL of our program into the testing script, but it has not been generated yet so the file does not exist.

```
import { HelloWorld } from "../target/types/hello_world";
```

IDL stands for Interface Description Language and they are a core concept of Anchor programs. We will cover them much more in depth later, but for now all you need to know is that an IDL defines how clients can talk to an Anchor program.

To test this program out, we are going to use the Anchor CLI. Specifically, we'll make use of the `anchor test` command. This is a very useful command in the Anchor CLI, it will compile the latest version of our program, update/create the program's IDL, deploy it to the configured cluster, and then run any integration tests we have written to send transactions to the contract. By default, the command is configured to start up a local validator on your machine and deploy your contract locally rather than to say Devnet, mainnet, or testnet.

> Since the `anchor test` command spins up a local validator, it will not work if you already have a validator running locally. If you run into issues with this command, it's always helpful to double check there are no other local validators running already in another terminal.

We can use the default testing code as is to test our hello world program. Just save your changes and run `anchor test` in the terminal. After it compiles and the test runs, you should see the transaction id printed to the console. You can see the log with our "Hello Metacrafters!" message inside the `.anchor` folder that was created, there should be a file with the program's logs there.

```
Transaction executed in slot 38: Signature:
4PMzfzSyV98Rxc56uy3Q6kR87eMNT4dCkALxKvE77daGBsCsmRDXKy5DzWC34v1ThL5kCf8bV4QkmEEBRy7hHm1
N Status: Ok Log Messages: Program Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS invoke
[1] Program log: Instruction: Initialize Program log: Hello Metacrafter! Program
Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS consumed 422 of 200000 compute units
Program Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS success
```

Summary

1. `declare_id!()` macro
2. the program module
3. the `Accounts` struct.

As a refresher, the `Accounts` struct is where you define which accounts your instruction expects and which constraints these accounts should adhere to. You do this via two constructs: types and constraints.

Accounts Types

When defining which accounts your program should expect to be passed in, Anchor has specific Account types that you can use to make your definitions even more specific. The types available to you are:

- `Account`
- `Signer`
- `Program`
- `SystemAccount`
- `Sysvar`
- `UncheckedAccount`
- `AccountInfo`
- `AccountLoader`

Each of these account types have a specific use case in mind.

Well, this is one way Anchor does that! Depending on which type you use to define the expected account, Anchor will automatically validate some aspects of the actual account that's passed in via a transaction.

For example, any account that you define with the `Signer` type, Anchor will verify that it has signed the incoming transaction. If it hasn't, then the program will return an error before the program logic is even executed!

The `Account` type is used when an instruction is interested in the deserialized data of the account. Anchor will inherently verify program ownership of any account defined with this type and deserialize its underlying data into a Rust type automatically. This means that when making use of this account type, you must also tell Anchor what to deserialize the data to.

Given the example below, we can see the `my_account` account is defined with `Account<'info, MyAccount>`.

```
pub my_account: Account<'info, MyAccount>,
```

The `Account` is generic over `T`. This `T` is a type you can create yourself to store data.

This tells Anchor to deserialize the raw byte data stored in whatever account that's passed in this field to the MyAccount data structure. You can create any number of customized data structs that you'd like to deserialize account data into.

```
#[account]
pub struct MyAccount {
    pub data: u64,
}
```

For those that don't know, deserialization is just the process of converting a byte buffer to a data structure or object. Serialization is the same idea, just in the opposite direction - the process of converting a data structure to a byte buffer.

Account requires `T` to implement certain functions (e.g. functions that (de)serialize `T`). Most of the time, you can use the `#[account]` attribute to add these functions to your data, as is done in the example.

This deserialization process is handled automatically behind the scenes by Anchor, but with Native Rust you'd have to implement this process manually. This is actually one of Anchor's biggest selling points to developers, as handling it manually is very tedious and non-trivial!

Most importantly, the `#[account]` attribute sets the owner of that data to the `ID` (the one we created earlier with `declare_id` macro!) of the crate `#[account]` is used in. The Account type can then check for you that the `AccountInfo` passed into your instruction has its `owner` field set to the correct program!

The `Signer` type validates that the account signed the transaction. No other ownership or type checks are done. If this is used, one should not try to access the underlying account data. Because the underlying data is not accessed or deserialized, there is no need to tell anchor which Rust type to deserialize it to when using this type definition.

The `Program` type validates that the account is the given Program. All the accounts that will be needed for the instruction are expected to be passed in every time, this includes any programs that the instruction will need access to. You can use this type to verify that the program passed in is the program you expect it to be. Just like you can tell Anchor what Rust data struct you'd like an account deserialized to, you can pass the program that you're expecting and Anchor will verify they match.

The `SystemProgram` type validates that the account is owned by the system program.

```
#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(init, payer = user, space = 8 + 8)]
    pub my_account: Account<'info, MyAccount>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct Update<'info> {
    #[account(mut)]
    pub my_account: Account<'info, MyAccount>,
}

#[account]
pub struct MyAccount {
    pub data: u64,
}
```

```
//! Account types that can be used in the account validation struct.

pub mod account; Account container that checks ownership on deserialization.

pub mod account_info; AccountInfo can be used as a type but [Unchecked Account]
(https://docs.rs/anchor-
lang/latest/anchor_lang/accounts/unchecked_account/struct.UncheckedAccount.html "struct
anchor_lang::accounts::unchecked_account::UncheckedAccount") should be used instead.
```

```
pub mod account_loader; Type facilitating on demand zero copy deserialization.
pub mod boxed; Box type to save stack space.

pub mod interface; Type validating that the account is one of a set of given Programs
pub mod interface_account; Account container that checks ownership on deserialization.
pub mod option; Option type for optional accounts.
pub mod program;
Type validating that the account is the given Program

pub mod signer; Type validating that the account signed the transaction
pub mod system_account; Type validating that the account is owned by the system program
pub mod sysvar; Type validating that the account is a sysvar and deserializing it
pub mod unchecked_account; Explicit wrapper for AccountInfo types to emphasize that no
checks are performed
```