Anchor Program Structure

TheAnchor framework usesRust macros to reduce boilerplate code and simplify the implementation of common security checks required for writing Solana programs.

The main macros found in an Anchor program include:

- declare_id
- : Specifies the program's on-chain address
- #[program]
- : Specifies the module containing the program’s
- instruction logic
- #[derive(Accounts)]
- : Applied to structs to
- indicate a list of accounts required by an instruction
- #[account]
- : Applied to structs to create custom account
- types for the program

Example Program#

Let's examine a simple program that demonstrates the usage of the macros mentioned above to understand the basic structure of an Anchor program.

The example program below creates a new account (NewAccount) that stores au64 value passed to theinitialize instruction.

lib.rs use anchor_lang :: prelude ::* ;

declare_id! ("11111111111111111111111111111111");

[program]

mod hello_anchor { use super ::* ; pub fn initialize (ctx : Context < Initialize

, data : u64) -> Result <()> { ctx . accounts . new_account . data = data; msg! ("Changed data to: {}!" , data); Ok (()) } }

[derive(

Accounts)] pub struct Initialize <' info

{

[account(init, payer

= signer, space = 8 + 8)] pub new_account : Account <' info , NewAccount

,

[account(

mut)] pub signer : Signer <' info

, pub system_program : Program <' info , System , }

[account]

pub struct NewAccount { data : u64 , }

declare_id! macro#

Thedeclare_id macro specifies the on-chain address of the program, known as the program ID.

lib.rs use anchor_lang :: prelude ::* ;

declare_id! ("11111111111111111111111111111111"); By default, the program ID is the public key of the keypair generated at/target/deploy/your_program_name.json .

To update the value of the program ID in thedeclare_id macro with the public key of the keypair in the/target/deploy/your_program_name.json file, run the following command:

Terminal anchor keys sync Theanchor keys sync command is useful to run when cloning a repository where the value of the program ID in a cloned repo'sdeclare_id macro won't match the one generated when you runanchor build locally.

#[program] macro#

The#[program] macro defines the module that contains all the instruction handlers for your program. Each public function within this module corresponds to an instruction that can be invoked.

lib.rs use anchor_lang :: prelude ::* ;

declare_id! ("11111111111111111111111111111111");

[program]

mod hello_anchor { use super ::* ; pub fn initialize (ctx : Context < Initialize

, data : u64) -> Result <()> { ctx . accounts . new_account . data = data; msg! ("Changed data to: {}!" , data); Ok (()) } }

[derive(

Accounts)] pub struct Initialize <' info

{

[account(init, payer

= signer, space = 8 + 8)] pub new_account : Account <' info , NewAccount

,

[account(

mut)] pub signer : Signer <' info

, pub system_program : Program <' info , System , }

[account]

pub struct NewAccount { data : u64 , }

Instruction Context#

Instruction handlers are functions that define the logic executed when an instruction is invoked. The first parameter of each handler is aContext type, whereT is a struct implementing theAccounts trait and specifies the accounts the instruction requires.

TheContext type provides the instruction with access to the following non-argument inputs:

pub struct Context <' a , ' b , ' c , ' info , T

{ /// Currently executing program id. pub program_id : & ' a Pubkey , /// Deserialized accounts. pub accounts : & ' b mut T , /// Remaining accounts given but not deserialized or validated. /// Be very careful when using this directly. pub remaining_accounts : & ' c [AccountInfo <' info], /// Bump seeds found during constraint validation. This is provided as a /// convenience so that handlers don't have to recalculate bump seeds or /// pass them in as arguments. pub bumps : BTreeMap < String , u8 , } TheContext fields can be accessed in an instruction using dot

notation:

- ctx.accounts
- : The accounts required for the instruction
- ctx.program_id
- : The program's public key (address)
- ctx.remaining_accounts
- : Additional accounts not specified in theAccounts
- struct.
- ctx.bumps
- : Bump seeds for anyProgram Derived Address (PDA)
- accounts specified in theAccounts
- struct

Additional parameters are optional and can be included to specify arguments that must be provided when the instruction is invoked.

lib.rs pub fn initialize (ctx :

Context < Initialize

, data : u64) -> Result <()> { ctx . accounts . new_account . data = data; msg! ("Changed data to: {}!" , data); Ok (()) } In this example, theInitialize struct implements theAccounts trait where each field in the struct represents an account required by theinitialize instruction.

lib.rs

[program]

mod hello_anchor { use super ::* ; pub fn initialize (ctx : Context < Initialize

, data : u64) -> Result <()> { ctx . accounts . new_account . data = data; msg! ("Changed data to: {}!" , data); Ok (()) } }

[derive(

Accounts)] pub struct

Initialize <' info

{

[account(init, payer

= signer, space = 8 + 8)] pub new_account : Account <' info , NewAccount

,

[account(

mut)] pub signer : Signer <' info

, pub system_program : Program <' info , System , }

#[derive(Accounts)] macro#

The#[derive(Accounts)] macro is applied to a struct to specify the accounts that must be provided when an instruction is invoked. This macro implements theAccounts trait, which simplifies account validation and serialization and deserialization of account data.

[derive(

Accounts)] pub struct Initialize <' info

{

[account(init, payer

= signer, space = 8 + 8)] pub new_account : Account <' info , NewAccount

,

[account(

mut)] pub signer : Signer <' info

, pub system_program : Program <' info , System , } Each field in the struct represents an account required by an instruction. The naming of each field is arbitrary, but it is recommended to use a descriptive name that indicates the purpose of the account.

[derive(

Accounts)] pub struct Initialize <' info

{

[account(init, payer

= signer, space = 8 + 8)] pub

new_account : Account <' info , NewAccount

,

[account(

mut)] pub

signer : Signer <' info

, pub

system_program : Program <' info , System

, }

Account Validation#

To prevent security vulnerabiliies, it's important to verify that accounts provided to an instruction are the expected accounts. Accounts are validated in Anchor programs in two ways that are generally used together:

- Account Constraints
- :
- Constraints define additional conditions that an account must satisfy to be
- considered valid for the instruction. Constraints are applied using the#[account(..)]
- attribute, which is placed above a field in a struct that
- implements theAccounts
- trait.
- You can find the implementation of the constraintshere
- .

- # [derive(

- Accounts
-)]
- pub
- struct

- Initialize
- <'
- info

- {

[account(init, payer

- =
- signer, space
- =
- 8
- +
- 8)]
- pub
- new_account
- :
- Account
- <'
- info
- ,
- NewAccount

- ,

[account(

- mut
-)]
- pub
- signer
- :
- Signer
- <'
- info

- ,

- pub
- system_program
- :
- Program
- <'
- info
- ,
- System

- ,

- }
- Account Types
- : Anchor
- provides various account types to help ensure that the account provided by the
- client matches what the program expects.
- You can find the implementation of the account types here
- .

[derive(

- Accounts
-)]
- pub
- struct
- Initialize
- <'
- info

- {

[account(init, payer

- =
- signer, space
- =
- 8
- +
- 8)]
- pub
- new_account
- :
- Account
- <'
- info
- ,
- NewAccount

- ,

[account(

- mut
-)]
- pub
- signer
- :
- Signer
- <'
- info

- ,

- pub
- system_program
- :
- Program
- <'
- info
- ,
- System

- ,

- }

When an instruction in an Anchor program is invoked, the program first validates the accounts provided before executing the instruction's logic. After validation, these accounts can be accessed within the instruction using thectx.accounts syntax.

lib.rs use anchor_lang :: prelude ::* ;

declare_id! ("11111111111111111111111111111111");

[program]

mod hello_anchor { use super ::* ; pub fn initialize (ctx : Context < Initialize

, data : u64) -> Result <()> { ctx . accounts . new_account . data = data; msg! ("Changed data to: {}!" , data); Ok (()) } }

[derive(

Accounts)] pub struct

Initialize <' info

{

[account(init, payer

= signer, space = 8 + 8)] pub

new_account : Account <' info , NewAccount

,

[account(

mut)] pub signer : Signer <' info

, pub system_program : Program <' info , System , }

[account]

pub struct NewAccount { data : u64 , }

#[account] macro#

The#[account] macro is applied to structs that define the data stored in custom accounts created by your program.

[account]

pub struct NewAccount { data : u64 , } This macro implements various traitsdetailed here . The key functionalities of the# [account] macro include:

- Assign Program Owner
- :
- When creating an account, the program owner of the account is automatically
- set to the program specified indeclare_id
- .
- Set Discriminator
- :
- A unique 8 byte discriminator, specific to the account type, is added as the
- first 8 bytes of account data during its initialization. This helps in
- differentiating account types and is used for account validation.
- Data Serialization and Deserialization
- :
- Account data is automatically serialized and deserialized as the account type.

lib.rs use anchor_lang :: prelude ::* ;

declare_id! ("11111111111111111111111111111111");

[program]

mod hello_anchor { use super ::* ; pub fn initialize (ctx : Context < Initialize

, data : u64) -> Result <()> { ctx . accounts . new_account . data

= data; msg! ("Changed data to: {}!" , data); Ok (()) } }

[derive(

Accounts)] pub struct Initialize <' info

{

[account(init, payer

= signer, space = 8 + 8)] pub new_account : Account <' info , NewAccount

,

[account(

mut)] pub signer : Signer <' info

, pub system_program : Program <' info , System , }

[account]

pub struct

NewAccount { data : u64 , }

Account Discriminator#

An account discriminator in an Anchor program refers to an 8 byte identifier unique to each account type. It's derived from the first 8 bytes of the SHA256 hash of the stringaccount: . This discriminator is stored as the first 8 bytes of account data when an account is created.

When creating an account in an Anchor program, 8 bytes must be allocated for the discriminator.

[account(init, payer

= signer, space =

8

+ 8)] pub new_account : Account <' info , NewAccount

, The discriminator is used during the following two scenarios:

- Initialization: When an account is created, the discriminator is set as the
- first 8 bytes of the account's data.
- Deserialization: When account data is deserialized, the first 8 bytes of
- account data is checked against the discriminator of the expected account
- type.

If there's a mismatch, it indicates that the client has provided an unexpected account. This mechanism serves as an account validation check in Anchor programs.

Previous «Anchor Framework Next IDL File»