IT'S TIME TO ENTER THE ARENA 📡 - SIGN UP FOR THE SOLANA RADAR HACKATHON NOW!

Documentation Courses Guides Cookbook Terminology
RPC API Stack Exchange

Home > Developers > Guides

Getting Started with the Anchor Framework

updated 24. April 2024

beginner rust anchor

The Anchor framework uses Rust macros to reduce boilerplate code and simplify the implementation of common security checks required for writing Solana programs.

Think of Anchor as a framework for Solana programs much like Next.js is for web development. Just as Next.js allows developers to create websites using React instead of relying solely on HTML and TypeScript, Anchor provides a set of tools and abstractions that make building Solana programs more intuitive and secure.

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 for an instruction
- `#[account]`: Applied to structs to create custom account types specific to the program

Anchor Program

Below is a simple Anchor program with a single instruction that creates a new account. We'll walk through it to explain the basic structure of an Anchor program. Here is the program on Solana Playground.

lib.rs

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

declare_id!("11111111111111111111111111111111");

```

```
#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>, d
        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

The `declare_id` macro is used to specify the on-chain address of the program (program ID).

lib.rs

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

```
declare_id!("11111111111111111111111111111111");
```

When you build an Anchor program for the first time, the framework generates a new keypair used to deploy the program (unless specified otherwise). The public key from this keypair should be used as the program ID in the `declare_id` macro.

- When using Solana Playground, the program ID is updated automatically for you and can be exported using the UI.
- When building locally, the program keypair can be found in `/target/deploy/your_program_name.json`

program macro

The `#[program]` macro specifies the module containing all of your program's instructions. Each public function in the module represents a separate instruction for the program.

In every function, the first parameter is always a `Context` type. Subsequent parameters, which are optional, define any additional `data` required by the instruction.

lib.rs

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

declare_id!("11111111111111111111111111111111");

#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>, d
        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,
}
```

The `Context` 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 deserial
    /// Be very careful when using this directly.
    pub remaining_accounts: &'c [AccountInfo<'inf
    /// Bump seeds found during constraint valida
    /// convenience so that handlers don't have t
    /// pass them in as arguments.
    pub bumps: BTreeMap<String, u8>,
}
```

`Context` is a generic type where `T` represents the set of accounts required by an instruction. When defining the instruction's `Context`, the `T` type is a struct that implements the `Accounts` trait (`Context<Initialize>`).

This context parameter allows the instruction to access:

- `ctx.accounts`: The instruction's accounts
- `ctx.program_id`: The address of the program itself
- `ctx.remaining_accounts`: All remaining accounts provided to the instruction but not specified in the `Accounts` struct
- `ctx.bumps`: Bump seeds for any Program Derived Address (PDA) accounts specified in the `Accounts` struct

derive(Accounts) macro

The `#[derive(Accounts)]` macro is applied to a struct and implements the `Accounts` trait. This is used to specify and validate a set of accounts required for a particular instruction.

```
#[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 that is 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>,
}
```

When building Solana programs, it's essential to validate the accounts provided by the client. This validation is achieved in Anchor through account constraints and specifying appropriate account types:

- 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 an account field in the `Accounts` struct.

```
#[derive(Accounts)]
```

```
pub struct Initialize<'info> {
    #[account(init, payer = signer, space = 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.

```
#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(init, payer = signer, space = 8
    pub new_account: Account<'info, NewAccou
    #[account(mut)]
    pub signer: Signer<'info>,
    pub system_program: Program<'info, Syster
}
```

Accounts within the `Accounts` struct are accessible in an instruction through the `Context`, using the `ctx.accounts` syntax.

lib.rs

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

declare_id!("11111111111111111111111111111111");

#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>,
        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,
}
```

When an instruction in an Anchor program is invoked, the program performs the following checks as specified the in `Accounts` struct:

- Account Type Verification: It verifies that the accounts passed into the instruction correspond to the account types defined in the instruction Context.
- Constraint Checks: It checks the accounts against any additional constraints specified.

This helps ensure that the accounts passed to the instruction from the client are valid. If any checks fail, then the instruction fails with an error before reaching the main logic of the instruction handler function.

For more detailed examples, refer to the constraints and account types sections in the Anchor documentation.

account macro

The `#[account]` macro is applied to structs to define the format of a custom data account type for a program. Each field in the struct represents a field that will be stored in the account data.

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

This macro implements various traits detailed here. The key functionalities of the `#[account]` macro include:

- Assign Ownership: When creating an account, the ownership of the account is automatically assigned to the program specified in the `declare_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 account validation.

- Data Serialization and Deserialization: The account data corresponding to the account type is automatically serialized and deserialized.

lib.rs

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

declare_id!("11111111111111111111111111111111");

#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>, d
        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,
}
```

In Anchor, an account discriminator is an 8-byte identifier, unique to each account type. This identifier is derived from the first 8 bytes of the SHA256 hash of the account type's name. The first 8 bytes in an account's data are specifically reserved for this discriminator.

```rust
#[account(init, payer = signer, space = 8 + 8)]
pub new_account: Account<'info, NewAccount>,
```

The discriminator is used during the following two scenarios:

- Initialization: During the initialization of an account, the discriminator is set with the account type's discriminator.
- Deserialization: When account data is deserialized, the discriminator within the data is checked against the expected discriminator of the 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, ensuring the correct and expected accounts are used.

IDL File

When an Anchor program is built, Anchor generates an interface description language (IDL) file representing the structure of the program. This IDL file provides a standardized JSON-based format for building program instructions and fetching program accounts.

Below are examples of how an IDL file relates to the program code.

Instructions

The `instructions` array in the IDL corresponds with the instructions on the program and specifies the required accounts and parameters for each instruction.

IDL.json

```
{
  "version": "0.1.0",
  "name": "hello_anchor",
  "instructions": [
    {
      "name": "initialize",
      "accounts": [
        { "name": "newAccount", "isMut": true, "i
        { "name": "signer", "isMut": true, "isSig
        { "name": "systemProgram", "isMut": false
      ],
      "args": [{ "name": "data", "type": "u64" }]
    }
  ],
  "accounts": [
    {
      "name": "NewAccount",
      "type": {
        "kind": "struct",
        "fields": [{ "name": "data", "type": "u64
      }
    }
  ]
}
```

lib.rs

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

declare_id!("11111111111111111111111111111111");

#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>, d
        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,
}
```

Accounts

The `accounts` array in the IDL corresponds with structs in the program annotated with the `#[account]` macro, which specifies the structure of the program's data accounts.

IDL.json

```
{
  "version": "0.1.0",
  "name": "hello_anchor",
  "instructions": [
    {
      "name": "initialize",
      "accounts": [
        { "name": "newAccount", "isMut": true, "i
        { "name": "signer", "isMut": true, "isSig
        { "name": "systemProgram", "isMut": false
      ],
      "args": [{ "name": "data", "type": "u64" }]
    }
  ],
  "accounts": [
    {
      "name": "NewAccount",
      "type": {
        "kind": "struct",
        "fields": [{ "name": "data", "type": "u64
      }
    }
  ]
}
```

lib.rs

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

declare_id!("11111111111111111111111111111111");

#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>, d
        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,
}
```

Client

Anchor provides a Typescript client library (`@coral-xyz/anchor`) that simplifies the process of interacting with Solana programs from the client.

To use the client library, you first need to set up an instance of a `Program` using the IDL file generated by Anchor.

Client Program

Creating an instance of the `Program` requires the program's IDL, its on-chain address (`programId`), and an `AnchorProvider` . An `AnchorProvider` combines two things:

- `Connection` - the connection to a Solana cluster (i.e. localhost, devnet, mainnet)
- `Wallet` - (optional) a default wallet used to pay and sign transactions

When building an Anchor program locally, the setup for creating an instance of the `Program` is done automatically in the test file. The IDL file can be found in the `/target` folder.

TS

```
import * as anchor from "@coral-xyz/anchor";
import { Program, BN } from "@coral-xyz/anchor";
import { HelloAnchor } from "../target/types/hell

const provider = anchor.AnchorProvider.env();
anchor.setProvider(provider);
const program = anchor.workspace.HelloAnchor as P
```

When integrating with a frontend using the wallet adapter, you'll need to manually set up the `AnchorProvider` and `Program`.

TS

```
import { Program, Idl, AnchorProvider, setProvide
import { useAnchorWallet, useConnection } from "@
import { IDL, HelloAnchor } from "./idl";

const { connection } = useConnection();
const wallet = useAnchorWallet();

const provider = new AnchorProvider(connection, w
setProvider(provider);

const programId = new PublicKey("...");
const program = new Program<HelloAnchor>(IDL, pro
```

Alternatively, you can create an instance of the `Program` using only the IDL and the `Connection` to a Solana cluster. This means if there is no default `Wallet`, but allows you to use the `Program` to fetch accounts before a wallet is connected.

TS

```
import { Program } from "@coral-xyz/anchor";
import { clusterApiUrl, Connection, PublicKey } f
import { IDL, HelloAnchor } from "./idl";

const programId = new PublicKey("...");
const connection = new Connection(clusterApiUrl("

const program = new Program<HelloAnchor>(IDL, pro
  connection,
});
```

Invoke Instructions

Once the `Program` is set up, you can use the Anchor `MethodsBuilder` to build an instruction, a transaction, or build and send a transaction. The basic format looks like this:

- `program.methods` - This is the builder API for creating instruction calls related to the program's IDL
- `.instructionName` - Specific instruction from the program IDL, passing in any instruction data as comma-separated values
- `.accounts` - Pass in the address of each account required by the instruction as specified in the IDL
- `.signers` - Optionally pass in an array of keypairs required as additional signers by the instruction

TS

```
await program.methods
  .instructionName(instructionData1, instructionD
  .accounts({})
  .signers([])
  .rpc();
```

Below are examples of how to invoke an instruction using the methods builder.

rpc()

The `rpc()` method sends a signed transaction with the specified instruction and returns a `TransactionSignature`. When using `.rpc`, the `Wallet` from the `Provider` is automatically included as a signer.

TS

```
// Generate keypair for the new account
const newAccountKp = new Keypair();

const data = new BN(42);
const transactionSignature = await program.method
  .initialize(data)
  .accounts({
    newAccount: newAccountKp.publicKey,
    signer: wallet.publicKey,
    systemProgram: SystemProgram.programId,
  })
  .signers([newAccountKp])
  .rpc();
```

transaction()

The `transaction()` method builds a `Transaction` and adds the specified instruction to the transaction (without automatically sending).

TS

```ts
// Generate keypair for the new account
const newAccountKp = new Keypair();

const data = new BN(42);
const transaction = await program.methods
  .initialize(data)
  .accounts({
    newAccount: newAccountKp.publicKey,
    signer: wallet.publicKey,
    systemProgram: SystemProgram.programId,
  })
  .transaction();

const transactionSignature = await connection.sen
  wallet.payer,
  newAccountKp,
]);
```

instruction()

The `instruction()` method builds a `TransactionInstruction` using the specified instruction. This is useful if you want to manually add the instruction to a transaction and combine it with other instructions.

TS

```ts
// Generate keypair for the new account
const newAccountKp = new Keypair();
const data = new BN(42);
const instruction = await program.methods
  .initialize(data)
  .accounts({
    newAccount: newAccountKp.publicKey,
    signer: wallet.publicKey,
    systemProgram: SystemProgram.programId,
  })
  .instruction();

const transaction = new Transaction().add(instru

const transactionSignature = await connection.sen
  wallet.payer,
  newAccountKp,
]);
```

Fetch Accounts

The client `Program` also allows you to easily fetch and filter program accounts. Simply use `program.account` and then specify the name of the account type on the IDL. Anchor then deserializes and returns all accounts as specified.

all()

Use `all()` to fetch all existing accounts for a specific account type.

TS

```
const accounts = await program.account.newAccount
```

memcmp

Use `memcmp` to filter for accounts storing data that matches a specific value at a specific offset. When calculating the offset, remember that the first 8 bytes are reserved for the account discriminator in accounts created through an Anchor program. Using `memcmp` requires you to understand the byte layout of the data field for the account type you are fetching.

```ts
const accounts = await program.account.newAccount
  {
    memcmp: {
      offset: 8,
      bytes: "",
    },
  },
]);
```

fetch()

Use `fetch()` to get the account data for a specific account by passing in the account address

```ts
const account = await program.account.newAccount.
```

fetchMultiple()

Use `fetchMultiple()` to get the account data for multiple accounts by passing in an array of account addresses

TS

```
const accounts = await program.account.newAccount
  ACCOUNT_ADDRESS_ONE,
  ACCOUNT_ADDRESS_TWO,
]);
```

Previous
« Create a token on Solana

Next
How to write a Native Rust Program »

Media Kit

Karriere

Haftungsausschluss

Privacy Policy

DE