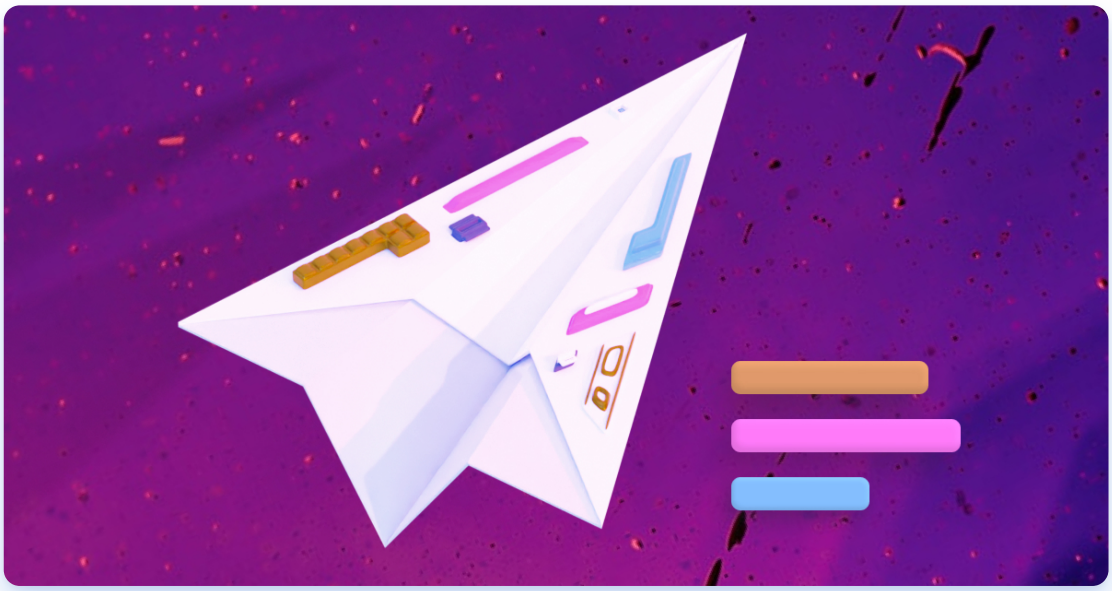
CREATE A SOLANA DAPP FROM SCRATCH

Our first instruction

EPISODE 4 1 YEAR AGO 14 MIN READ



Now that our Tweet account is defined and ready to be used, let's implement an instruction that allows users to send their own tweets.

Defining the context

As we've seen in the previous episode, programs are special accounts that store their own code but cannot store any other information. We say that **programs in Solana are stateless**.

Because of that, sending an instruction to a program requires providing all the necessary context for it to run successfully.

Similarly to how we defined our `Tweet` account, contexts are implemented using a `struct`. Within that `struct`, we should list all the accounts that are necessary for the instruction to do its job.

In your `lib.rs` file, just above the `Tweet` struct we defined in the previous episode, you should see an empty `Initialize` context.

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

Let's replace that `Initialize` context with a `SendTweet` context and list all the accounts we need in there.

Remove the two lines above and replace them with the following code.

```rust
#[derive(Accounts)]
pub struct SendTweet<'info> {
    pub tweet: Account<'info, Tweet>,
    pub author: Signer<'info>,
    pub system_program: AccountInfo<'info>,
}
```

There's a bunch of new stuff here so I'll first focus on the accounts themselves and then explain a few Rust features that might look confusing.

First of all, adding an account on a context simply means **its public key should be provided when sending the instruction**.

Additionally, we might also require the account to **use its private key to sign the instruction** depending on what we're planning to do with the account. For instance, we will want the `author` account to sign the instruction to ensure somebody is not tweeting on behalf of someone else.

Okay, let's have a quick look through the listed accounts:

- `tweet` : This is the account that the instruction will create. You might be wondering why we are giving an account to an instruction if that instruction creates it. The answer is simple: we're simply passing the public key that should be used when creating the account. We'll also need to sign using its private key to tell the instruction we own the public key. Essentially, we're telling the instruction: "here's a public key that I own, create a Tweet account there for me please".
- `author` : As mentioned above, we need to know who is sending the tweet and we need their signature to prove it.
- `system_program` : This is the official System Program from Solana. As you can see, because programs are stateless, we even need to pass through the official System Program. This program will be used to initialize the `Tweet` account and figure out how much money we need to be rent-exempt.

Next, let's explain some of Rust's quirks we can see in the code above.

- `#[derive(Accounts)]` : This is a derive attribute provided by Anchor that allows the framework to generate a lot of code and macros for our `struct` context. Without it, these few lines of code would be a lot more complex.
- `<'info>` : This is a Rust lifetime. It is defined as a generic type but it is not a type. Its purpose is to tell the Rust compiler how long a variable will stay alive for.

Rest assured, there's no need to dig deeper into these Rust features to follow this series. I'm just throwing some references for the interested readers.

Finally, let's talk about types. Each of these properties has a different type of account so what's up with that? Well, they all represent an account but with slight variations.

- `AccountInfo` : This is a low-level Solana structure that can represent any account. When using `AccountInfo` , the account's data will be an unparsed array of bytes.
- `Account` : This is an account type provided by Anchor. It wraps the `AccountInfo` in another `struct` that parses the data according to an account `struct` provided as a generic type. In the example above, `Account<'info, Tweet>` means this is an account of type `Tweet` and the data should be parsed accordingly.
- `Signer` : This is the same as the `AccountInfo` type except we're also saying this account should sign the instruction.

Note that, if we want to ensure that an account of type `Account` is a signer, we can do this using account constraints.

Account constraints

On top of helping us define instruction contexts in just a few lines of code, Anchor also provides us with account constraints that can be defined as Rust attributes on our account properties.

Not only these constraints can help us with security and access control, but they can also help us initialise an account for us at the right size.

This sounds perfect for our `tweet` property since we're creating a new account in this instruction. For it to work, simply add the following line on top of the `tweet` property.

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

    pub tweet: Account<'info, Tweet>,
        pub author: Signer<'info>,
        pub system_program: AccountInfo<'info>,
    }
```

However, the code above will throw an error because we are not telling Anchor how much storage our `Tweet` account needs and who should pay for the rent-exempt money. Fortunately, we can use the `payer` and `space` arguments for that purpose.

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

        pub tweet: Account<'info, Tweet>,
        pub author: Signer<'info>,
        pub system_program: AccountInfo<'info>,
    }
```

The `payer` argument references the `author` account within the same context and the `space` argument uses the `Tweet::LEN` constant we defined in the previous episode. Isn't it amazing that we can do all of that in just one line of code?

Now, because we're saying that the `author` should pay for the rent-exempt money of the `tweet` account, we need to mark the `author` property as mutable. That's because we are going to mutate the amount of money in their account. Again, Anchor makes this super easy for us with the `mut` account constraint.

```
    #[derive(Accounts)]
    pub struct SendTweet<'info> {
        #[account(init, payer = author, space = Tweet::LEN)]
        pub tweet: Account<'info, Tweet>,
        pub author: Signer<'info>,
        pub system_program: AccountInfo<'info>,
    }
```

Note that there's also a `signer` account constraint that we could use on the `author` property to make sure they have signed the instruction but it is redundant in our case because we're already using the `Signer` account type.

Finally, we need a constraint on the `system_program` to ensure it really is the official System Program from Solana. Otherwise, nothing stops users from providing us with a malicious System Program.

To achieve this, we can use the `address` account constraint which requires the public key of the account to exactly match a provided public key.

```
    #[derive(Accounts)]
    pub struct SendTweet<'info> {
        #[account(init, payer = author, space = Tweet::LEN)]
        pub tweet: Account<'info, Tweet>,
        #[account(mut)]
        pub author: Signer<'info>,

        pub system_program: AccountInfo<'info>,
    }
```

The `system_program::ID` is a constant defined in Solana's codebase. By default, it's not included in Anchor's `prelude::*` import so we need to add the following line afterwards — at the very top of our `lib.rs` file.

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

EDIT 2022-03-22: In newer versions of Anchor, we can achieve the same result by using yet another type of account called `Program` and passing it the `System` type to ensure it is the official System program.

```
#[derive(Accounts)]
pub struct SendTweet<'info> {
    // ...
    pub system_program: Program<'info, System>,
}
```

And just like that, we're done with defining the context of our `SendTweet` instruction.

Note that Anchor provides a lot more constraints for us. Click here and scroll down a bit to see the exhaustive list of constraints it supports.

Implementing the logic

Now that our context is ready, let's implement the actual logic of our `SendTweet` instruction.

Inside the `solana_twitter` module, replace the `initialize` function with the following code.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    Ok(())
}
```

A few things to note here:

- We've renamed the `initialize` instruction to `send_tweet`. Function names are snake cased in Rust.
- We've replaced the generic type inside `Context` to `SendTweet` to link the instruction with the context we created above.
- We've added two additional arguments: `topic` and `content`. Any argument which is not an account can be provided this way, after the context.
- This function returns a `ProgramResult` which can either be `Ok` or `ProgramError`. Rust does not have the concept of exceptions. Instead, you need to wrap your return value into a special enum to tell the program if the execution was successful (`Ok`) or not (`Err` and more specifically here `ProgramError`). Since we're not doing anything inside that function for now, we immediately return `Ok(())` which is an `Ok` type with no return value inside `()`. Also, note that the last line of a function is used as the return value without the need for a `return` keyword.

Now that our function signature is ready, let's extract all the accounts we will need from the context.

First, we need to access the `tweet` account which has already been initialised by Anchor thanks to the `init` account constraint. You can think of account constraints as middleware that occur before the instruction function is being executed.

We can access the `tweet` account via `ctx.accounts.tweet`. Because we're using Rust, we also need to prefix this with `&` to access the account by reference and `mut` to make sure we're allowed to mutate its data.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content


    Ok(())
}
```

Similarly, we need to access the `author` account to save it on the `tweet` account. Here, we don't need `mut` because Anchor already took care of the rent-exempt payment.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    let tweet: &mut Account<Tweet> = &mut ctx.accounts.tweet;


    Ok(())
}
```

Finally, we need access to Solana's `Clock` system variable to figure out the current timestamp and store it on the tweet. That system variable is accessible via `Clock::get()` and can only work if the System Program is provided as an account.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    let tweet: &mut Account<Tweet> = &mut ctx.accounts.tweet;
    let author: &Signer = &ctx.accounts.author;


    Ok(())
}
```

Note that we're using the `unwrap()` function because `Clock::get()` returns a `Result` which can be `Ok` or `Err`. Unwrapping a result means either using the value inside `Ok` — in our case, the clock — or immediately returning the error.

Including the `topic` and the `content` passed as arguments, We now have all the data we need to fill our new `tweet` account with the right data.

Let's start with the author's public key. We can access it via `author.key` but this contains a reference to the public key so we need to dereference it using `*`.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    let tweet: &mut Account<Tweet> = &mut ctx.accounts.tweet;
    let author: &Signer = &ctx.accounts.author;
    let clock: Clock = Clock::get().unwrap();



    Ok(())
}
```

Then, we can retrieve the current UNIX timestamp from the clock by using `clock.unix_timestamp`.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    let tweet: &mut Account<Tweet> = &mut ctx.accounts.tweet;
    let author: &Signer = &ctx.accounts.author;
    let clock: Clock = Clock::get().unwrap();

    tweet.author = *author.key;


    Ok(())
}
```

Finally, we can store the `topic` and the `content` in their respective properties.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    let tweet: &mut Account<Tweet> = &mut ctx.accounts.tweet;
    let author: &Signer = &ctx.accounts.author;
    let clock: Clock = Clock::get().unwrap();

    tweet.author = *author.key;
    tweet.timestamp = clock.unix_timestamp;



    Ok(())
}
```

At this point, we have a working instruction that initialises a new `Tweet` account for us and hydrates it with the right information.

Guarding against invalid data

Whilst Anchor's account constraints protect us from lots of invalid scenarios, we still need to make sure our program reject data that's not valid from our own requirements.

In the previous episode, we decided to use the `String` type for both the `topic` and the `content` properties and allocate 50 characters max for the former and 280 characters max for the latter.

Since the `String` type is a vector type and has no fixed limit, we haven't made any restrictions on the number of characters the topic and the content can have. We've only allocated the right amount of storage for them.

Currently, nothing could stop a user from defining a topic of 280 characters and a content of 50 characters. Even worse, since most characters only need one byte to encode and nothing forces us to enter a topic, we could have a content that is (280 + 50) * 4 = 1320 characters long.

Therefore, if we want to protect ourselves from these scenarios, we need to add a few guards.

Let's add a couple of if statements before hydrating our `tweet` account. We'll check that the `topic` and the `content` arguments aren't more than 50 and 280 characters long respectively. We can access the amount of characters a `String` contains via `my_string.chars().count()`. Notice how we're not using `my_string.len()` which returns the length of the vector and therefore gives us the number of bytes in the string.

```rust
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    let tweet: &mut Account<Tweet> = &mut ctx.accounts.tweet;
    let author: &Signer = &ctx.accounts.author;
    let clock: Clock = Clock::get().unwrap();









    tweet.author = *author.key;
    tweet.timestamp = clock.unix_timestamp;
    tweet.topic = topic;
    tweet.content = content;
    Ok(())
}
```

Now that the if statements are in place, we need to return an error inside them to stop the execution of the instruction early.

Anchor makes dealing with errors a breeze by allowing us to define an `ErrorCode` enum using the `#[error_code]` Rust attribute. For each type of error inside the enum, we can provide a `#[msg("...")]` attribute that explains it.

Let's implement our own `ErrorCode` enum and define two errors inside of it. One for when the topic is too long and one for when the content is too long.

You can copy/paste the following code at the end of your `lib.rs` file.

```
#[error_code]
pub enum ErrorCode {
    #[msg("The provided topic should be 50 characters long maximu
    TopicTooLong,
    #[msg("The provided content should be 280 characters long max
    ContentTooLong,
}
```

Now, let's use the errors we've just defined inside our if statements.

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String, content
    let tweet: &mut Account<Tweet> = &mut ctx.accounts.tweet;
    let author: &Signer = &ctx.accounts.author;
    let clock: Clock = Clock::get().unwrap();
```

```
            return Err(ErrorCode::TopicTooLong.into())
        }


            return Err(ErrorCode::ContentTooLong.into())
        }

        tweet.author = *author.key;
        tweet.timestamp = clock.unix_timestamp;
        tweet.topic = topic;
        tweet.content = content;
        Ok(())
    }
```

As you can see, we first need to access the error type like a constant — e.g. `ErrorCode::TopicTooLong` — and wrap it inside an `Err` enum type. The `into()` method is a Rust feature that converts our `ErrorCode` type into whatever type is required by the code which here is `Err` and more precisely `ProgramError`.

Awesome, not only we're protected against invalid topic and content sizes but we also know how to add more error types and guards in the future.

Instruction vs transaction

Before wrapping up this article, I'd like to mention the difference between an instruction and a transaction because they are commonly used interchangeably and it did bug me at first.

The difference is simple though: **a transaction is composed of one or multiple instructions**.

When a user interacts with the Solana blockchain, they can push many instructions in an array and send all of them as one transaction. The benefit of this is that **transactions are atomic**, meaning that if any of the instructions fail, the entire operation rolls back and it's like nothing ever happened.

Instructions can also delegate to other instructions either within the same program or outside of the current program. The latter is called Cross-Program Invocations (CPI) and the signers of the current instruction are automatically passed along to the nested instructions. Anchor even has a helpful API for making CPI calls.

No matter how many instructions and nested instructions exists inside a transaction, it will always be atomic — i.e. it's all or nothing.

Whilst we haven't and we won't directly use multiple and nested instructions per transaction in this series, we have used them indirectly already. When using the `init` account constraint from Anchor, we asked Anchor to initialise a new account for us and it did this by calling the `create_account` instruction of Solana's System Program — therefore making a CPI.

Before we close this long parenthesis, it can be useful to know that instructions are often abbreviated `ix` whilst transactions are often abbreviated `tx`.

Conclusion

Believe it or not, our Solana program is finished! 🥳

We've defined our `Tweet` account and implemented an instruction that creates new ones on demand and stores all relevant information. As usual, you can find the code for that episode on GitHub.

View Episode 4 on GitHub

Compare with Episode 3

From this point forward, we will focus on *using* our program by interacting with its instruction and fetching existing accounts.

Eventually, we'll do this in a fully-fledged JavaScript application but first, we'll do this in tests to make sure everything is working properly. See you in the next episode!

Next episode →

Testing our instruction

← Previous episode

Structuring our Tweet account

Discussions

solana_primat 11 months ago

Perfect. Thank you for this tutorials.

💖 0 0 replies

Jose 10 months ago

Oh man, thank you so much for writing this in better English. Rust is tough, and programming in Solana without too much Rust knowledge (even after reading the boo and coming from high level programming languages it's still tough.

I'm going to be opening a GitHub issue. Just for future readers. This won't compile in Anchor 0.22.1.

You'd need to replace the `#[error]` macro with `#[error_code]` .

Replace the `ProgramResult` for `Result` return type from the pub fn tweet like this:

```
pub fn send_tweet(ctx: Context<SendTweet>, topic: String,
```

Then, return the error with the `error!` macro:

```
if topic.chars().count() > 50 {
        return Err(error!(ErrorCode::TopicTooLong));
    }

    if content.chars().count() > 280 {
        return Err(error!(ErrorCode::ContentTooLong));
    }
```

In the end, you will have:

```
#[error_code]
pub enum ErrorCode {
    #[msg("The provided topic should be 50 characters long ma
    TopicTooLong,
    #[msg("The provided content should be 280 characters long
    ContentTooLong,
}

pub fn send_tweet(ctx: Context<SendTweet>, topic: String, con
        let tweet: &mut Account<Tweet> = &mut ctx.accounts.tw
        let author: &Signer = &ctx.accounts.author;
        let clock: Clock = Clock::get()?;

        if topic.chars().count() > 50 {
            return Err(error!(ErrorCode::TopicTooLong));
        }

        if content.chars().count() > 280 {
            return Err(error!(ErrorCode::ContentTooLong));
        }
        tweet.author = *author.key;
        tweet.timestamp = clock.unix_timestamp;
        tweet.topic = topic;
        tweet.content = content;

        Ok(())
    }
```

For more info ,see here: https://project-serum.github.io/anchor/tutorials/tutorial-4.html#defining-a-program

💖 7 1 reply

Hakeemullah J. Yousufzai 10 months ago

Thnks Jose for the solution

💖 2 0 replies

Challenger 7 months ago

Your post is very helpful for me.. Best regards.!!

💖 0 0 replies

Gábor Kovács 2 months ago

for the tweets can't we just use a PDA as well?

💖 0 0 replies

Would you like to chime in?

You must be a member to start a new discussion.

Fortunately, it only takes two click to become one. See you on the other side! 🌸

Become a Member