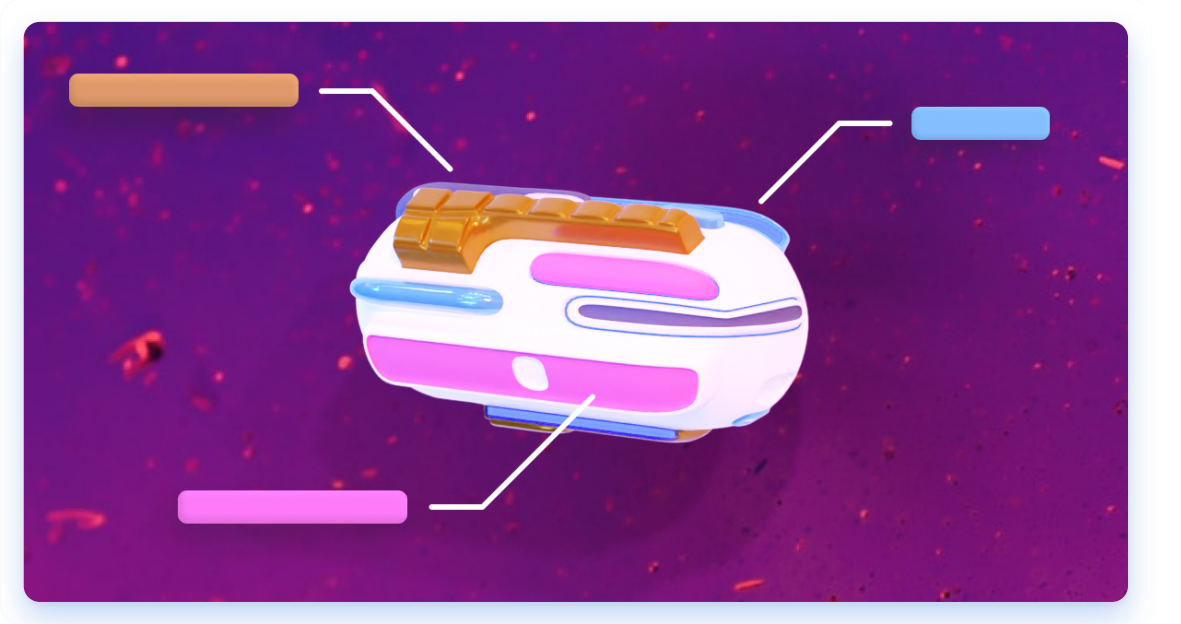
CREATE A SOLANA DAPP FROM SCRATCH

Structuring our Tweet account

EPISODE 3 1 YEAR AGO 12 MIN READ



Accounts are the building blocks of Solana. In this episode, we'll explain what they are and how to define them in our programs.

Everything is an account

In Solana, **everything is an account**.

This is a fundamental concept that differs from most other blockchains. For instance, if you've ever created a smart contract in Solidity, then you ended up with a bunch of code that can store a bunch of data and interact with it. Any user that interacts with a smart contract ends up updating data inside a smart contract. That's not the case in Solana.

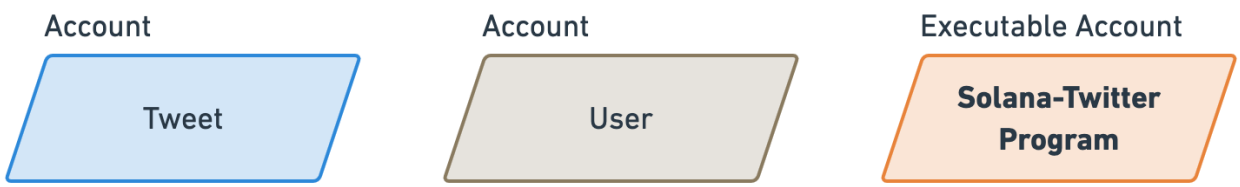
In Solana, if you want to store data somewhere, **you've got to create a new account**. You can think of accounts as little clouds of data that store information for you in the blockchain. You can have one big account that stores all the information you need, or you can have many little accounts that store more granular information.

Programs may create, retrieve, update or delete accounts but they need accounts to store information as it cannot be stored directly in the program.

But here is where it becomes more interesting: **even programs are accounts**.

Programs are special accounts that store their own code, are read-only and are marked as "executable". There's literally an `executable` boolean on every single account that tells us if this account is a program or a regular account that stores data.

So remember, everything is an account in Solana. Programs, Wallets, NFTs, Tweets, there're all made of accounts.



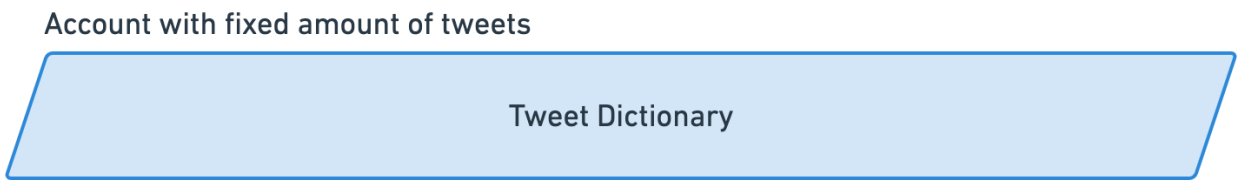
Defining an account for our Tweets

Okay, let's define our first account. We need to define a structure that will hold all of the information we need to publish and display tweets.

Solution A 👎

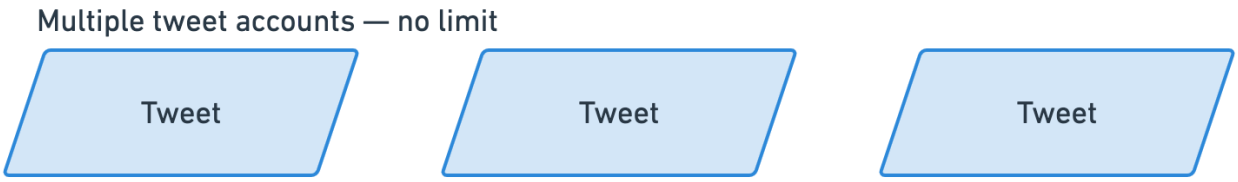
We could have one big account storing all of the tweets ever created but that wouldn't be a very scalable solution because we need to allocate a fixed size to our accounts. That means we would need to define a maximum number of tweets allowed to be published by everyone.

Additionally, someone has to pay for the storage that will exist on the blockchain. If we have to pre-allocate the storage for every single tweet, we will end up paying for everybody's storage. And the more storage we require, the more expensive it will be.



Solution B 👍

A better solution would be to have **every tweet stored on its own account**. That way, storage will be created and paid on demand by the author of the tweet. Since each tweet will require only a small amount of space, the storage will be more affordable and will scale to an unlimited amount of tweets and users. Granularity pays in Solana.



Implementation

Let's implement solution B in our Solana program. Open your `lib.rs` file, that's where we'll implement the entirety of our program. You can ignore all the existing code for now and add the following at the end of the file.

```
// programs/solana-twitter/src/lib.rs

// ...

#[account]
pub struct Tweet {
    pub author: Pubkey,
    pub timestamp: i64,
    pub topic: String,
    pub content: String,
}
```

That's it! These 7 lines of codes are all we needed to define our tweet account. Now, time for some explanations.

`#[account]` . This line is a custom Rust attribute provided by the Anchor framework. It removes a huge amount of boilerplate for us when it comes to defining accounts — such as parsing the account to and from an array of bytes. Thanks to Anchor, we don't need to know much more than that so let's be grateful for it.

`pub struct Tweet`. This is a Rust struct that defines the properties of our Tweet. If you're not familiar with structs (in Rust, C or other languages), you can think of them as classes that only define properties (no methods).

`author`. We keep track of the user that published the tweet by storing its public key.

`timestamp`. We keep track of the time the tweet by published by storing the current timestamp.

`topic`. We keep track of an optional "topic" field that can be provided by the user so their tweet can appear on that topic's page. Twitter does that differently by parsing all the hashtags (#) from the tweet's content but that would be pretty challenging to achieve in Solana so we'll extract that to a different field for the sake of simplicity.

`content`. Finally, we keep track of the actual content of the tweet.

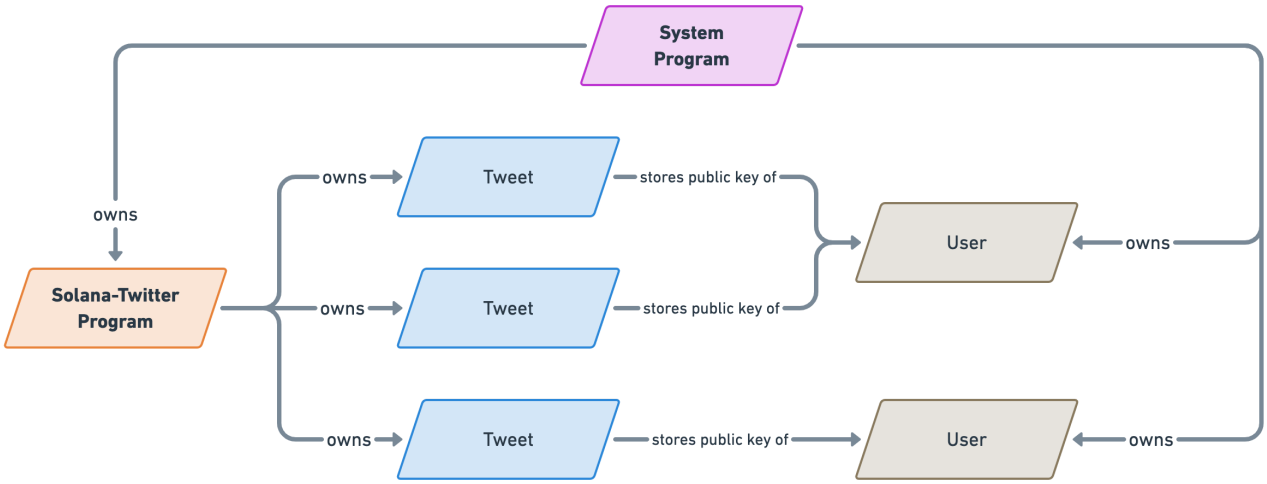
Why store the author?

You might think that creating an account on the Solana blockchain keeps track of its owner, and you'd be right! So why do need to keep track of the Tweet's author inside the account's data?

That's because **the owner of an account will be the program that generated it**.

Therefore, if we didn't store the public key of the author that created the tweet, we'd have no way of displaying the author later on, and even worse, we'd have no way of allowing that user — and that user only — to perform actions such as updating or deleting their own tweets.

Here's a little diagram that shows how all accounts will be related to one another.



As you can see even our "solana-twitter" program is owned by another account which is Solana's System Program. This executable account also owns every user account. The System Program is ultimately the ancestor of all Solana accounts.

An incentive to store less

Okay, now that we know what we want to store in our `Tweet` account, we need to define the total size of our account in bytes. We will need to know that size on the next episode when we create our first instruction that will send tweets to the blockchain.

Technically, we don't have to provide the optimal size to store our data. We could simply tell Solana that we want our Tweet accounts to be, say, 4000 bytes (4kB). That should be more than enough to store all our content. So why don't we? Because Solana gives us an incentive not to.

Rent

Rent is an important concept in Solana and ensures everybody that adds data to the blockchain is accountable for the amount of storage they provide.

The concept is simple:

> When an account is created, someone has to put some money into it.
>
> Every so often, the blockchain collects some of that money as a “rent”. That rent is proportional to the size of the account.
>
> When the account runs out of money, the account is deleted and your data is lost!

Wow, wait what?!

Yes, if you’re account cannot pay the rent at the next collection, it will be deleted from the blockchain. But don’t panic, that does not mean we are destined to pay rent on all of our tweets for the rest of our days. Fortunately, there is a way to be rent-exempt.

Rent-exempt

In practice, everybody creates accounts that are rent-exempt, meaning rent will not be collected and the account will not risk being deleted. Ever.

So how does one create a rent-exempt account? Simple: you need to **add enough money in the account to pay the equivalent of two years of rent**.

Once you do, the money will stay on the account forever and will never be collected. Even better, if you decide to close the account in the future, **you will get back the rent-exempt money**!

Solana provides Rust, JavaScript and CLI tools to figure out how much money needs to be added to an account for it to be rent-exempt based on its size. For example, run this in your terminal to find out the rent-exempt minimum for a 4kB account.

```
# Ensure your local ledger is running for this to work.
solana rent 4000

# Outputs:
# Rent per byte-year: 0.00000348 SOL
```

```
# Rent per epoch: 0.000078662 SOL
# Rent-exempt minimum: 0.02873088 SOL
```

That being said, we won't be needing these methods in our program since Anchor takes care of all of the math for us. All we need to figure out is how much storage we need for our account, so let's do that now.

Sizing our account

Earlier, we defined our Tweet account with the following properties:

- `author` of type `PubKey`.
- `timestamp` of type `i64`.
- `topic` of type `String`.
- `content` of type `String`.

Therefore, to size our account, we need to figure out **how many bytes** each of these properties require and sum it all up.

But first, there's a little something you should know.

Discriminator

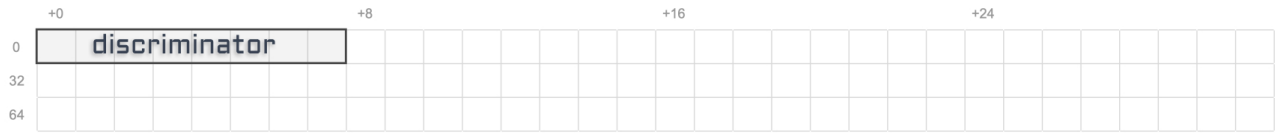
Whenever a new account is created, **a `discriminator` of exactly 8 bytes will be added** to the very beginning of the data.

That discriminator stores the type of the account. This way, if we have multiple types of accounts — say a Tweet account and a UserProfile account — then our program can differentiate them.

Alright, let's keep track of that information in our code by adding the following constant at the end of the `lib.rs` file.

```
const DISCRIMINATOR_LENGTH: usize = 8;
```

Also, if you're a visual person like me, here's a little representation of the storage we've established so far where each cell represents a byte.



Author

Good, now we can move on to our actual properties, stating with the author's public key.

How do we find out the size of the `PubKey` type? If you're using an IDE such as CLion, you can control-click on the `PubKey` type and it will take you to its definition. Here's what you should see.

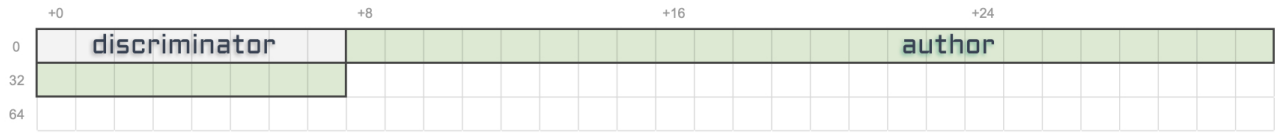
```
pub struct Pubkey([u8; 32]);
```

This special looking struct defines an array. The size of each item is given in the first element and the length of the array is given in the second element. Therefore, that struct defines an array of 32 items of type `u8`. The type `u8` means it's an unsigned integer of 8 bits. Since there are 8 bits in one byte, we end up with a total array length of **32 bytes**.

That means, to store the `author` property — or any public key — we only need 32 bytes. Let's also keep track of that information in a constant.

```
const PUBLIC_KEY_LENGTH: usize = 32;
```

And here is our updated storage representation.

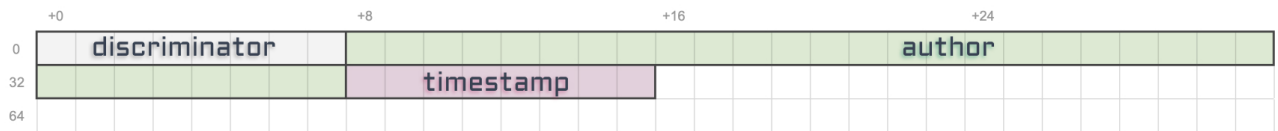


Timestamp

The timestamp property is of type `i64`. That means it's an integer of 64 bits or **8 bytes**.

Let's add a constant, see our updated storage representation and move on to the next property.

```
const TIMESTAMP_LENGTH: usize = 8;
```



Topic

The topic property is a bit more tricky. If you control-click on the `String` type, you should see the following definition.

```
pub struct String {
    vec: Vec<u8>,
}
```

This struct defines a vector (`vec`) containing elements of 1 byte (`u8`). A vector is like an array whose total length is unknown. We can always add to the end of a vector as long as we have enough storage for it.

That's all nice but how do we figure its storage size if it's got no limit?

Well, that depends on what we intend to store in that `String` . We need to explicitly figure out what we want to store and what is the maximum amount of bytes it could require.

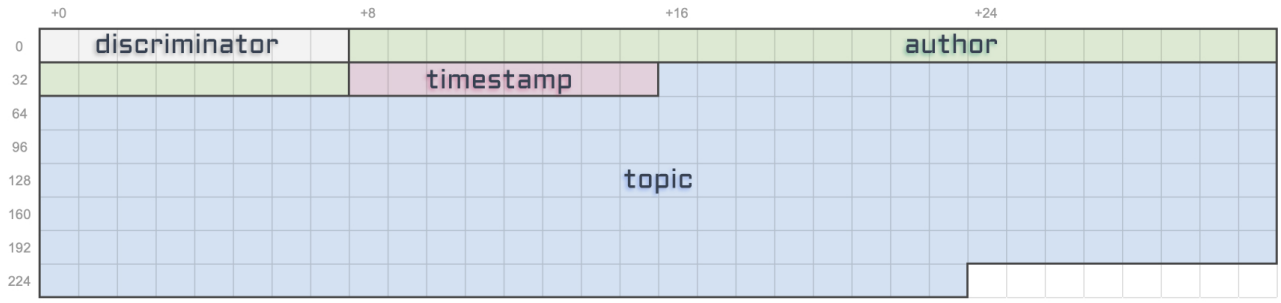
In our case, we're storing a topic. That could be: `solana` , `laravel` , `accessibility` , etc.

So let's make a decision that a topic will have a **maximum size of 50 characters**. That should be enough for most topics out there.

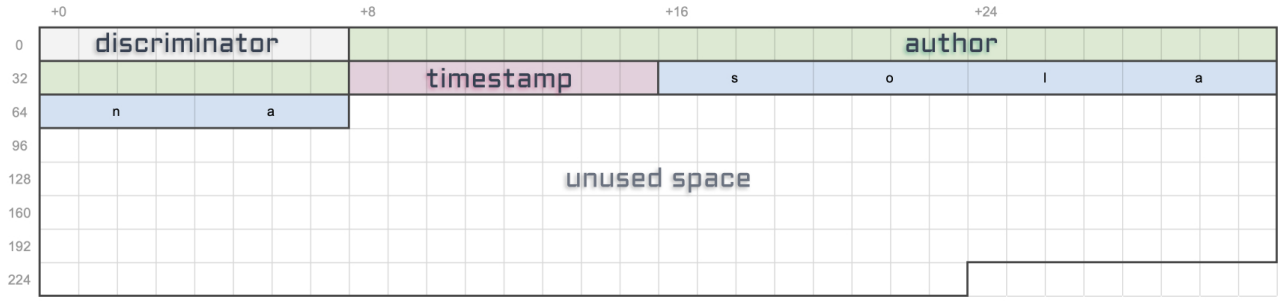
Now we need to figure out how many bytes are required to store one character.

It turns out, using UTF-8 encoding, a character can use from 1 to 4 bytes. Since we need the maximum amount of bytes a topic could require, we've got to size our characters at 4 bytes each.

Okay, so far we have figured out that our topic property should at most require 50 x 4 = **200 bytes**.



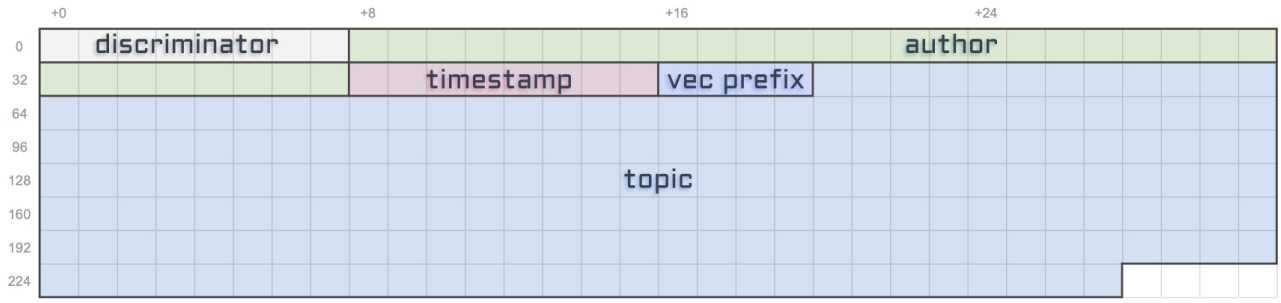
It’s important to note that this size is purely indicative since vectors don’t have limits. So whilst we’re allocating for 200 bytes, typing “solana” as a topic will only require 6 x 4 = 24 bytes.



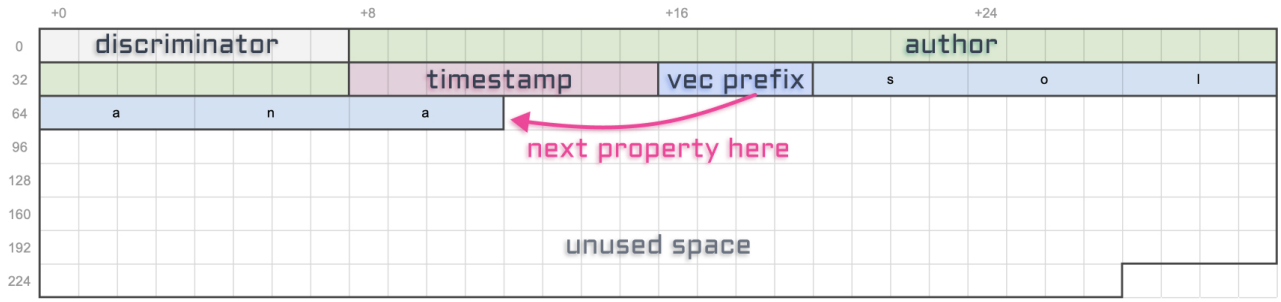
Note that characters in “solana” don’t require 4 bytes but I’m pretending for simplicity.

We’re almost done with our topic property but there’s one last thing to think about when it comes to the `String` type or vectors in general.

Before storing the actual content of our string, there will be a **4 bytes prefix** whose entire purpose is to store its total length. Not the maximum length that it could be, but the actual length of the string based on its content.



That prefix is important to know where the next property is located on the array of bytes. Since vectors have no limits, without that prefix we wouldn't know where it stops.



Phew! Okay, now that we know how to size `String` properties, let's define a few constants that summarise our findings.

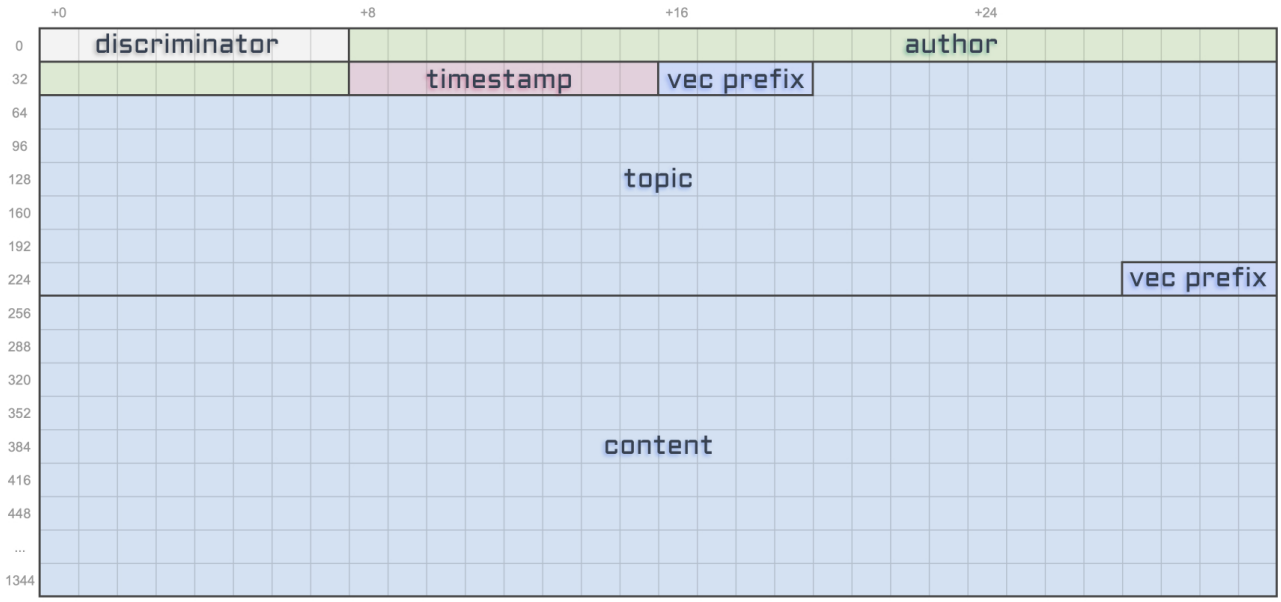
```
const STRING_LENGTH_PREFIX: usize = 4; // Stores the size of the
const MAX_TOPIC_LENGTH: usize = 50 * 4; // 50 chars max.
```

Content

We've already done all the hard work of understanding how to size `String` properties so this will be super easy.

The only thing that differs from the topic property is the character count. Here, we want the content of our tweets to be a **maximum of 280 characters** which make the total size

of our content 4 + 280 * 4 = **1124 bytes**.



As usual, let's add a constant for this.

```
const MAX_CONTENT_LENGTH: usize = 280 * 4; // 280 chars max.
```

Size recap

Sizing properties can be hard on Solana so here's a little recap table that you can refer back to when sizing your accounts.

If anyone would like to add to this table, feel free to reach out to me and I'll make sure to keep this up-to-date.

EDIT 2022-03-24: There's now a similar table in the official Anchor Book called "Space References". Be sure to check it out. 🙂

Type	Size	Explanation
`bool`	1 byte	1 bit rounded up to 1 byte.
`u8` or `i8`	1 byte	
`u16` or `i16`	2 bytes	
`u32` or `i32`	4 bytes	
`u64` or `i64`	8 bytes	
`u128` or `i128`	16 bytes	
`[u16; 32]`	64 bytes	32 items x 2 bytes. `[itemSize; arrayLength]`
`PubKey`	32 bytes	Same as `[u8; 32]`
`vec<u16>`	Any multiple of 2 bytes + 4 bytes for the prefix	Need to allocate the maximum amount of item that could be required.
`String`	Any multiple of 1 byte + 4 bytes for the prefix	Same as `vec<u8>`

Final code

Let's have a look at all the code we've written in this article and combine our various constants into one that gives the total size of our Tweet account.

```
// 1. Define the structure of the Tweet account.
#[account]
pub struct Tweet {
    pub author: Pubkey,
    pub timestamp: i64,
    pub topic: String,
    pub content: String,
}
```

```
// 2. Add some useful constants for sizing propeties.
const DISCRIMINATOR_LENGTH: usize = 8;
const PUBLIC_KEY_LENGTH: usize = 32;
const TIMESTAMP_LENGTH: usize = 8;
const STRING_LENGTH_PREFIX: usize = 4; // Stores the size of the
const MAX_TOPIC_LENGTH: usize = 50 * 4; // 50 chars max.
const MAX_CONTENT_LENGTH: usize = 280 * 4; // 280 chars max.

// 3. Add a constant on the Tweet account that provides its total
impl Tweet {
    const LEN: usize = DISCRIMINATOR_LENGTH
        + PUBLIC_KEY_LENGTH // Author.
        + TIMESTAMP_LENGTH // Timestamp.
        + STRING_LENGTH_PREFIX + MAX_TOPIC_LENGTH // Topic.
        + STRING_LENGTH_PREFIX + MAX_CONTENT_LENGTH; // Content.
}
```

The third section of this code defines an implementation block on the `Tweet` struct. In Rust, that's how we can attach methods, constants and more to structs and, therefore, make them more like classes.

In this `impl` block, we define a `LEN` constant that simply sums up all the previous constants of this episode. That way we can access the length of the Tweet account in bytes by running `Tweet::LEN`.

And we're done with this episode! 😚

Conclusion

Even though we didn't write that much code, we saw why and how Solana gives us an incentive to think twice about the amount of storage we push to the blockchain.

We also took the time to understand how each property can be sized into an optimal amount of byte and defined reusable constants for better readability.

As usual, you can find the code for this episode on the `episode-3` branch of the repository.

View Episode 3 on GitHub

Compare with Episode 2

In the next episode, we will add more code to our `lib.rs` file to create our first instruction which will be responsible for creating a new Tweet account.

Next episode →

Our first instruction

← Previous episode

Getting started with Solana and Anchor

Discussions

Ahmed Ali 1 year ago

Thank you for the amazing series! can't stress how useful it is. Even after reading and fiddling around Solana and Anchor documentation, I am still finding this series helpful and putting things together quite nicely!

Thank you for all the efforts! One question/suggestion: I believe defining the system_account as AccountInfo with address constraint is equivalent to: `pub system_program: Program<'info, System>,` Am I right?

💖 0 1 reply

Dominique 1 year ago

Thank you very much. Your articles are very didactic.

💖 1 0 replies

Rixiao Zhang 10 months ago

This is amazing!!!

💖 1 0 replies

Rixiao Zhang 10 months ago

This is amazing!!!

💖 1 0 replies

Rahul 10 months ago

This is pure class. Great technical documentation.

💖 1 0 replies

markjames 10 months ago

I read your articles very seriously.

💖 1 0 replies

Hakeemullah J. Yousufzai 10 months ago

I love reading these. Outstanding

💖 1 0 replies

Leon 9 months ago

The "space reference" link of Anchor doc is updated now: https://book.anchor-lang.com/anchor_references/space.html

💖 0 0 replies

Chr1s 5 months ago

Awesome!

💖 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