- Welcome to the full end-to-end Solana Blockchain

and Smart Contract Bootcamp.

We're gonna go through everything you need to know

from a technical standpoint to get started building

on the Solana blockchain

and write your own full stack decentralized applications.

Whether or not you have been in the cryptocurrency space

for a while or just getting started,

we'll be taking you through your journey to learn

how to become an expert Solana engineer.

At the time of recording,

Solana developers are in high demand.

We're seeing a ton of jobs getting posted

within the Solana Ecosystem.

Over the last three years, Solana developers participating

in global hackathons have created companies

that have received over $600 million in funding.

And on average, Blockchain Developers right now

are making more than $170,000 a year.

While we cannot guarantee you a job,

we're gonna try our best to provide you

with the skills you need to become a Solana developer.

Whether or not you have any blockchain experience,

or have been building for a while,

if you are interested in building in blockchain,

this is the course for you.

We'll be starting with blockchain basics,

then going into full end-to-end projects,

and finally finishing up with what it takes

to get your new application to production.

If you have already built some smart contracts

or are familiar with blockchain,

feel free to jump around the bootcamp

to find which projects best fit your needs.

You'll be able to find all the information you need to know

about each project in the description,

as well as in the GitHub repository link below.

You'll find the entire bootcamp in this GitHub repository,

including any possible resources you may need,

such as links to documentation or further reading on topics.

Throughout this bootcamp, we'll be learning

how to build on a blockchain and be using both Rust

and TypeScript to accomplish this.

Don't worry if you don't know these languages

as we'll be teaching you enough to get started

throughout each lesson.

As long as you have some software development background,

you should do well.

Even the best developers don't get it right the first time.

If you do happen to run into issues at any point

in this bootcamp, there's a Solana specific stack exchange

where you can get help.

The Solana Foundation developer relations team

and other developers from the ecosystem will be answering

your questions there.

Just make sure you search for your question first

and provide enough information

about your error you've received

while building each project.

We will also post the full solution at the end of each step

and in the notes below.

It is incredibly important that you pace yourself

while going through this Bootcamp.

This is a marathon, not a sprint.

Take breaks so you can have some time to digest what you have just learned.

Go back to different projects as a refresher,

take the course at your own pace.

My name is Jacob Creech, and I'll be working with you

on some of these projects throughout this bootcamp.

- I'm Brianna Migliaccio, and I'll be leading you through

how to build Smart Contracts,

and Decentralized Applications.

- And I'm Mike MacCana.

I'll be working with you throughout the bootcamp

on key blockchain concepts.

Before you start building, you should understand the purpose

of blockchains and the problems that they solve.

Fully understanding the value blockchain provides

will help you leverage it to solve the right problems.

So let's get started.

So what is blockchain?

Why is it important and what can you do with it?

At its core, a blockchain is a distributed database

with them special characteristics.

Unlike most databases, blockchains are unique in that they have no central authority,

are fully open for anyone to use,

and all data is completely transparent.

Everyone can make payments on the blockchain

while no one entity is capable of stopping

or reversing those payments.

This is what we call censorship resistant.

The transactions are irreversible and the blockchain itself

is a recording of those transactions.

That's why blockchains are often referred to as a ledger.

Unlike databases which are typically stored

in a central location, data on blockchains are stored

on multiple computers across the world.

Essentially, there's no single copy of the blockchain

that can be lost or tampered with.

Instead, the data is stored in multiple copies and verified

by checking the blocks or bundles of transactions.

This is called decentralization.

What this means in practice is that blockchain allows people

to transact with each other directly.

Think about the last time you paid for something,

or swapped currencies or just stored your money somewhere.

You might have used a credit card,

or a money transfer service or a bank.

To facilitate that,

there's typically a service fee involved.

The seller pays a couple of percent

on every credit card transaction.

You pay a little bit more for an international transfer.

Annual bank probably has a monthly fee

just to hold your money.

Blockchain allows people to transact with each other

directly without needing somebody in between.

Now, to understand blockchain,

you need to start with Bitcoin.

Back in 2008, Bitcoin was created

by the pseudonym, Satoshi Nakamoto as a way

to make digital peer-to-peer payments.

The white paper for Bitcoin outlined a way

to make those payments without having a trusted third party

such as a bank or other financial institution.

Blockchain uses digital signatures to let people transact

without having to fully trust each other.

Digital signatures give us confidence that the money sent

is in the correct currency, that the amount is correct

and the amount was actually sent.

Bitcoin was created more than 16 years ago,

and these days is less used for day-to-day transactions.

People think of it more as digital gold.

However, plenty of new innovations

have been built since then.

Modern blockchains like the one that we'll be building on,

Solana support fast payments with lower fees

than both Bitcoin and traditional finance.

Not only that, but new creations like Smart Contracts

have enabled a whole new set of capabilities

previously not available to blockchains.

Smart Contracts are computer programs

that run on a blockchain.

Instead of having a cloud provider run your program,

you run the program in the blockchain.

Smart Contracts are a little bit like APIs

in the traditional tech world,

except rather than being invoked by requests,

they're invoked by instructions.

And rather than sending responses back,

they write their changes out to the blockchain

where anyone can read.

Smart contracts extend blockchains beyond simple payments

to allow complex transactions that could lend money

without a bank, allow people to exchange items directly

with each other at a price they both agree on,

or even create a lottery without a lottery ticket issuer.

We're going to build these in this bootcamp, so you can see

how powerful smart contracts can be for yourself.

Blockchain would not be possible without cryptography.

Everyone that uses a blockchain

has something called a key pair.

A key pair consists of two keys,

a private key and a public key.

The public key is shown to anyone.

It's used as an address people can use to transact with you.

The private key you must keep secret.

The private key is used to sign transactions,

which proves that you made them.

For example, wanna spend some of your tokens?

You'll need to sign the transaction using your private key.

Anyone else can use your public key

to verify that you as the holder

of the matching private key made that transaction.

Key pairs are pretty common in cryptography,

even outside blockchain.

Let's use the example of passports.

Your electronic passport is signed

by your government using your government's private key.

When you pass through the passport gates of the airport,

they use your government's public key

to prove your passport was really signed by your government.

If the passport signature is correct,

it must have been signed with your government's private key.

If I make my own passport, it won't be signed

by my government's private key,

so I won't be able to get through the gates.

In Solana, everyone has a public key

that uses their address.

Other people can use your address to transact with you.

The most basic Solana wallet is just an address

with a small balance of SOL.

Users can pay their transaction fees

and interact with smart contracts using SOL.

Here's Alice's wallet, here's Bob's wallet.

If Alice wanted to send Bob some SOL,

she would use her private key to sign a transaction

that uses the system program's transfer function

to send some SOL to Bob

and uses the memo programs memo function to write a note

to Bob about the transaction.

If both these steps called instructions

are completed successfully,

the transaction completes successfully

and Bob receives the tokens in his address.

If something goes wrong, that's okay.

The transaction fails and nothing changes,

Alice will still have her SOL.

Blockchain technology allows for new applications

to be built in this digital age.

We can create systems where people transact

with each other directly make digital assets

that can be sold or used outside of a game, allow art to be sold

and ensure artists get paid each time a piece of artwork

is transfered.

The projects we'll be building throughout this course

will use the Solana blockchain to create unique applications

that could only be built with smart contracts.

Let's get started with your first application.

The first smart contract we make will be simple.

It's gonna save our favorite things to the blockchain.

We're gonna learn how we can save, update,

and retrieve information from the blockchain and how signing

is used to control access in our smart contract.

In the previous lesson,

we learned how everyone has a key pair,

a public key that can be used as an address to transact

with that person and a private key that they use

to sign transactions.

In Solana, your smart contracts

or programs can store additional data

in what's called Program Derived Addresses.

PDA addresses are not made from a public key,

instead they're made from seeds,

which can be whatever you the programmer want.

Wanna store some config for your smart contracts?

You can make a PDA from the seed config as a string.

When you need to find your program config,

look for the PDA made from that seed

and you'll find your config.

Wanna store Alice's review of the movie "Titanic?"

You can combine Alice's wallet address

and the string "Titanic" to store the movie review.

Need to store Bob's review of the same film.

You can combine Bob's wallet address

and the string "Titanic" to store his review.

If you've used a key value store before,

you might be thinking this is familiar.

And yes, Solana PDAs are a type of key value store.

If you haven't used a key value store before,

you can think of a PDA as a role of data,

but the seeds being the primary key used to find the data.

So let's build a simple app

that saves someone's favorite thanks to the blockchain

and uses digital signatures to ensure only the wallet holder can update their own favorites.

We'll use Solana Playground, and Anchor,

most popular framework for making Solana programs.

So we're gonna be using a tool called Solana Playground,

which lets us make Solana programs

or smart contracts directly in our web browser

without installing anything on our local machine.

So open beta.solpg.io, click create a new project.

Give your project a name.

I'm gonna call this project favorites

because it's gonna save our favorite things

to the blockchain and we'll pick the Anchor option

and click create.

There's a file made for you,

but let's actually delete everything inside that file,

that lib.rs file.

We're gonna make everything from scratch ourselves.

So the first thing you'll see

in pretty much any Anchor program

is importing what's called the Anchor prelude,

which is the thing you'll see at the top of almost any file

and allows us to access everything from Anchor just by typing its name rather than typing out

the full path to it.

So use anchor lang prelude star,

that pulls in all of our prelude into the current scope.

The double colon is just Rust separator for namespace.

Other languages would use a dot for the same thing.

A program has a program Id also called an address.

I'm gonna set up a program address for our smart contract.

In Solana Playground,

we don't actually need to fill this in.

It will be done for us when we deploy our program.

Also just for neatness,

I'm gonna make a little constant here

and then I'm gonna call it ANCHOR DISCRIMINATOR SIZE.

The Anchor Discriminator is something that's written

to every account on the blockchain by an Anchor program

that just specifies the type of account it is.

And that's used by Anchor for some of its checks.

So that is a usize and the value is just eight.

So when we save things to the blockchain,

we'll need eight bytes plus the size

of whatever we're saving.

You'll see that later on.

Now the great thing about Anchor

is that we can take a regular Rust program

and with a particular macro turn it into an Anchor program.

So here is just a regular Rust module,

pub, mod favorites, and it's going to use

all those good things we just imported a moment ago.

And it has a function.

So hub, function, set favorites,

which is going to return. Nothing.

Oops, return nothing. And here is its body.

So this is just a regular Rust module.

It's called Favorites.

It has some functions inside,

in this case just one function set favorites.

But the great thing about Anchor

is that if we do this macro program,

suddenly that favorites Rust module

becomes a full Solana program or smart contract.

This is the power of Anchor.

It saves you from having to do boring things and also applies a bunch of sensible safe defaults.

So set favorites,

that function is now a Solana instruction handler.

It will take instructions that are provided by clients,

and run them and then save the results to the blockchain.

So let's think about what we're actually gonna save

to the blockchain.

We're gonna save people's favorite things.

So that's gonna be a struct called Favorites.

And inside we're gonna save people's favorite number,

which will be a 64 bit or eight byte number.

And we will have a favorite color,

which will just be a string.

And we'll have a list of hobbies,

which will be a vector of strings.

So pub hobbies, vector,

it's a little bit like an array of strings.

Now 'cause we're gonna save this to an account,

we'll also put the account macro on it.

And to let Anchor know how big Favorites is,

we're also gonna put this derive InitSpace.

That gives all of our instances of Favorites

that the InitSpace attribute,

which is the total space used by all the items inside.

We'll also need to specify the size of each individual item

to get that.

Obviously you know strings could be any size at all.

So we'll need to specify max length.

That's max len, say 50 bytes and I'll copy that for hobbies.

Hobbies is nested 'cause we have a vector

of a certain amount of items and then strings

which have a certain size.

So for hobbies we'll say five and 50 for the vector

and the string respectively.

When people call our set Favorites function,

they're gonna need to provide a list of accounts

that they're gonna modify on the blockchain.

One of the things that makes Solana great

is that if there's a bunch of people over here

who are running a transaction involving their accounts

and there's a bunch of other people over here running

a different transaction involving their accounts,

those transactions don't block each other.

There's no overlap in the accounts evolved.

So Solana can process them at the same time without waiting

for the other one to be finished.

So we'll make a struct to store this information

and we need to come up with a good name for it.

Now this is the struct of accounts

for our set Favorites instruction handler.

So the tradition is to call this struct of accounts

the same thing as our set Favorites instruction handler

just in title case.

Structs in Rust have title case and the convention

is to name the struct the same thing

as the function itself just in title case.

We could call this set Favorites accounts

but it's good to get along with other people.

And this is the convention.

So pub user, we're gonna need people to supply an account

that we call the user and that's gonna need to be a signer.

You'll see me doing this info stuff here.

All that is, is saying these items will live

for the lifetime of a Solana account info object.

This is just Rust having rules

about how long things should be kept in memory.

This isn't a Rust course,

I'm not gonna go into Lifetimes fully,

but we'll throw some links in the course descriptions,

so you can learn a little bit more about Lifetimes

if you want to.

So we're gonna specify some options for this account.

I'm gonna say that the signer is mutable. Why?

Because the person that signs the instruction

to set Favorites is going to pay to create

their favorites account on the blockchain.

You're gonna pay to store that information.

Also, we'll need the person running set Favorites

to specify the Favorites account they want to write to.

It doesn't mean that we'll let them write to this account,

we just need them to specify the account.

So we'll call that pub favorites.

And the type is, it's an account,

it goes for the lifetime of info and it is...

This an account which is an account of this Favorites struct

that we made earlier.

We're gonna add a few Anchor account options

as well to Favorites.

The first thing we're gonna do is specify init if needed.

Init if needed says make the favorites account

if it doesn't already exist.

We'll specify who pays to create the account,

which as we said before will be the user, this person here.

And that means the user is the person

who signed the transaction.

So the person who signed the transaction will be the person

who pays to create the Favorites account.

We're gonna tell Solana how much space this account needs.

So every account in Anchor has eight bytes

or the Anchor discriminator size using

that constant we made earlier,

plus the whatever the size for this struct.

So when we made our favorite struct,

which specified what favorites look like, their types

and the different keys that we need,

we said derive init space

that gave us this great init space trait, which we are using to calculate the size we will need

for a Favorites account.

In other words, a Favorites account is just an account

that has an Anchor discriminator and stores favorites.

We'll need to have some seeds.

Now the seeds are what's used to give this account

an address on the blockchain.

This is a program derived address unlike regular user

accounts, this isn't a public key.

The address for this is actually made based on some seeds

that we provide.

So if ever you used a key value store,

this is a really similar concept.

The keys in this case are whatever we have as our seeds.

And I'm gonna use, in this case the text favorites as bytes

and just the user's own key as well.

This means that if I'm storing my favorites, I will store

that under the address made from favorites as bytes

and my own public key.

And if I'm storing your favorites,

I will store that under an address made from favorites

as bytes and your public key.

So what we're doing here is storing information

under different accounts that are found based

on the inputs we give and the inputs we give

are based on the users.

Finally, we'll also specify the bump,

which is just used to calculate those seeds.

We have pub favorites and it's account, I'll type favorites.

That looks good.

The last thing we'll need to...

The last account we'll need people to specify

is just the system program.

The system program is used for so many things in Solana.

It's not the system program, it's the token program.

So that will be a program.

It lasts for the lifetime of info,

and it is a program of type system.

That looks good.

So at this point we have our favorites struct,

which is what we're gonna write into the blockchain

for every user.

And we have our set favorites list of accounts that people need to provide, which is also a struct.

Let's go and fill in our set favorites function,

which is our actual instruction handler.

The thing that people are gonna call from the instructions

in their Solana transactions.

We can do...

Let's do a little greeting message.

Solana has this message macro that's built into it,

or Anchor actually has this message macro

that's built into it.

We can do things like greetings from,

and we can say context.

So message is kind of like console log or print line.

It just writes things out to the Solana log file.

If you use Solana Explorer,

you'll actually be able to see these messages

when someone has a transaction that makes a call

to this instruction handler.

Let's also make a little variable just to show

the user's public key,

the person who's calling our instruction handler,

user public key equals context.accounts.user.key.

That's good. Let's add another log message

that just logs out the options

that people have provided. User.

Now I just realized that we needed to provide some arguments

to set favorites and we haven't done that yet.

So let's do that now.

Context. The first argument

to our set Favorites instruction handler

will be a context which is a context of Set Favorites.

In other words the Set Favorites accounts

that we made earlier.

And I will spell favorites using American English,

so I'm consistent.

We'll also ask people for all the options they wanna save.

So that's the number, that U64,

the color which is a string.

And hobbies, which is a vector of strings.

Save and let Solana playground do its formatting

and that looks good.

So yeah, our Set Favorites has a context,

a number of color and hobbies,

and you can see that we can access information

about like what program we're part of,

or all the accounts that people have specified through

that context parameter.

So back to our message, we have user public key

and we're gonna log that their favorite color is,

or their favorite number.

Favorite color is color,

and their hobbies are hobbies.

And just because hobbies is a vector of strings,

we're gonna add a colon and a question mark so it expands

the items inside that vector of strings.

The last thing we need to do

inside our Set Favorites function

is go and actually write the information

into the favorites account provided.

So that is context.accounts.favorites

and if we do a set inner, we can write information

into that account and the information will be right...

Will be an instance of the Favorites struct

with the number, the color and the hobbies

that have been provided.

That looks good.

Finally we will return just with okay and nothing,

that empty brackets and we don't need a semicolon

on that last line.

We emit the semicolon and Rust returns

the actual okay response.

This is how Solana instruction handlers work.

They write information to the blockchain

rather than returning it.

So that's our completed Anchor program.

We import the Prelude,

we set up the ID or program address.

I like to make an ANCHOR DISCRIMINATOR SIZE

rather than adding eight later on

'cause we know that's how much space

that every Anchor account needs at minimum.

We have a Rust module that has functions inside

and we apply this Solana program macro

that turns that Rust module into an Anchor program.

And then inside that module each function

becomes an Anchor instruction handler.

Each instruction handler has a context

which is all the accounts that the people calling

that instruction have provided plus any other arguments

for the instruction.

We're gonna log a couple of things and then we're gonna save

the information somebody has provided

to the favorites accounts,

we return okay just to let calling programs know

that this instruction was completed successfully.

If all the instructions in a transaction complete,

then the transaction is considered complete and everything

is written out to the blockchain.

If anything fails then nothing changes.

And we have two Structs, we have our favorite struct

which defines what we want to write out to the blockchain,

in this case a number of color and hobbies.

And also we're gonna add this great InitSpace macro,

so we can easily calculate how much space it will take

to store that.

And finally we have our accounts struct,

which I actually realized we need to add derive accounts

to the top of so that Anchor knows

this is our account struct and that specifies the accounts

that people need to provide along with the instruction.

So in this case it's just the user, the favorites account

and the system program.

One of the things that is great about this program

is that it already ensures that the person that is signing

the program has to be writing

to their own favorites account.

And we'll test this in a sec,

but I wanna show you the logic here.

In our account struct we say we have an account called

the user who is whoever signed the transaction.

And in our favorites, we say that favorites accounts,

the seeds that are used to find this address are made

from favorites and that user's key.

So if Alice signs this transaction,

and tries to write to Bob's account,

it won't actually work because the address

of the favorites account

won't actually match her user account.

We'll see this in a second, but it's a good example of one of the things that Anchor provides

for us out of the box in terms of smart, safe defaults.

Cool. So that looks good.

I'm going to build this program.

So I'm gonna slide this little do hickey at the bottom up.

So let's click the build button and build our program.

We might have some errors, we'll find out.

Yep. And we do.

It looks like I made a typo

'cause I'm not very good at talking

and typing at the same time.

I meant to return a result,

instead I wrote the word return.

So let's go and fix that.

That returns a result of nothing. There we go.

And let's build again.

There's another error.

Let's go have a look at the first one.

I can see that's really obvious.

It says on line 44 we have derived accounts

and that should actually be a capital.

So if we build once more,

you can see the syntax highlighting actually makes it look

a little bit better as well for the accounts.

You can see that completes successfully. That's great.

Now we will need to deploy.

When we go and deploy,

Anchor will actually go and make a key pair for us.

We can save this if we want to.

I'm not gonna use this again, so I'm not going to,

but maybe you should save the key pair.

And it says we don't have enough SOL

to complete the deployment.

I'm not gonna use...

I'm not gonna request an airdrop,

instead I am actually going to use the Solana faucet

which has some more features to get airdrops

than the inbuilt Solana playground airdrop tool.

An airdrop by the way is just a way

of getting free Devnet Sol.

So if I open up that address,

in Solana Explorer you can see that it has no balance.

It's a balance of SOL is accounted does not exist, data size is zero, et cetera.

So if I go and copy that address

and then go to faucets.solana.com,

paste in my wallet address

and let's ask for five SOL. Why not?

And you'll see the message Airdrop was successful.

If I go back into Solana Explorer you will see

that I now have five SOL. Excellent. Great.

So let's go and I'll clear that those old log messages

and let's hit deploy again back in Solana playground.

Good. So it's deploying

and it says that it can take a while, which it does.

And you'll notice while it deploys,

it actually takes a little bit of my SOL

to do that deployment and it's finished successfully.

So let's go check this out on Solana Explorer.

There's actually a transaction for deploying my program

and you can see that transaction was successful,

my deployment was successful.

Good things. Excellent.

So back here we have our Solana program,

and it's been deployed on Solana Devnet at the address.

You'll see by the way

that Solana playground actually added our program's deployed

address inside declare id, and in Solana playground.

We can actually go and search for that address

and find our program there.

It has some executable data, it's upgradeable,

there's the upgrade authority, which is the temporary key

that Solana Playground made for us,

in my case it starts with F. Great.

Solana Playground also has a test tab.

It's this little test tube on the left.

And this allows us to actually invoke

any of our instruction handlers on the blockchain straight

from Solana Playground.

So in this case I can go and set the favorites

for the temporary wallet that Solana Playground made for me.

I'm gonna specify my favorite number, which is seven,

make it custom, color which is red. Custom.

And Hobbies, which is skiing and oops, bike riding.

That looks good. Custom.

So as well as our favorite things,

people calling this instruction handler will also need

to specify the accounts that they want to use.

So for the user, pick the current wallet,

for the favorites pick from seed

and the first seed will be what we had before,

which is the string of the word favorites.

There we go.

Oops, sorry, I need to hit that string,

custom, favorites. There we go.

I'll add a seed, which is just the current wallet

and that matches what we had before

where we're calculating the seeds

to find this favorites account from those inputs,

the word favorites and the current user's wallet.

Now our program actually checks

that the favorites account we write to belongs

to the signer.

And I'll show you what happens

if someone else tries to write there later on.

The program is already written for us,

it's just the public key of our actual program

and I think that's it.

So we can go, oh let's make that C,

click that generate button and if we hit the test button,

we've actually made a transaction to our program.

So let's open that up on Solana Explorer.

So in Solana Playground here is our transaction

and it had only one instruction,

which is the call to our set favorites.

And if you look in the logs here,

you can actually see some really interesting things.

I'll make this font a little bit bigger as well.

So you can see that.

It says greetings from our program's public key.

Our user, our wallet address,

their favorite number is seven,

their favorite color is red and their hobbies are skiing

and bike riding.

And you can see that, that's actually saved that out

to that account.

The way you can do that by the way,

is you click on favorites and if you just do fetch ball,

you can actually see that in Solana Explorer we have a fully public key of our user,

we have a number of color and their set of hobbies.

Now what happens if a another user tries

to write to our favorites?

So back here if I pick for user, if I get rid of that

and I replace current wallet with random,

but I'm gonna leave in the favorites account

that was made from the previous user.

So if I test that now, if I try and run a transaction

where I'm writing to someone else's favorites account

from a random wallet,

you will see that a seeds constraint was violated

and the instruction did not complete successfully.

This is one of the great things about Anchor

is that these controls over what accounts people are allowed

to write to, are, you know, really handled for us.

Most of it was, you know this logic that we set up inside

the set favorites account struct

where we said that the user is whoever signs the transaction

and the favorites account must be made from these seeds.

If they specified a different favorites account

that was not made from those seeds,

that's where we get a seeds constraint was violated.

So you've now made a program, deployed it to Solana Devnet

and seen how Anchors account constraints allow us

to control access to data on the blockchain.

You've also, by the way written data to the blockchain

and read it back.

In the next steps, we're going to install Solana

on your own machine and create more programs

that do more useful things.

See you soon.

- The next application we're going to build

is all about voting.

Traditionally democratic governments hold elections,

and collect votes from the population.

There's a multitude of ways that people can vote,

but verification is still a very human process.

Votes are added to a ledger, and after a long period of time

the results will be released.

Using blockchain technology will not only be able

to immediately calculate the votes, but will also be able

to instantly verify a user voting using key pairs.

In this project you'll learn how to manage accounts,

and build using program derived addresses

with best practices.

Let's get started.

- Hello, my name is Jacob Creech,

and we're going to be building the voting application.

Before we get started on that,

we have to build up our local environment

to actually build our voting application.

This is contrary to the previous application

that you built, the Favorites app.

We're going to build it locally

versus all within the Solana Playground.

So let's get started

on how you set up your local environment.

All right, so first thing that you need to do

is you'll need to install Rust.

And the webpage that I'm looking at right here

is this is the Anchor docs for installation

and has a full list

of all the different things you need to install.

So first thing we're gonna do is install Rust, and go to this website and we just use this command.

So let's go type it into our command prompt and let it run.

So this will just make sure everything

is set up on our computer to use Rust to compile Anchor,

and other Solana smart contracts.

All right, so this is gonna update my Rust Home variable.

It's just giving me all the information

to do the installation, and so I want to just proceed

with standard installation and here we go.

Alright, so now we've had Rust installed

and the next thing we do as it says is we have to make sure

that we reload the path variable.

So we can see that here, we need to make sure

that to configure current shell you need to source

the corresponding environment variable under Home Cargo.

It's usually done by Cargo Environment,

or Cargo Environment Fish depending on what you're using.

You'll have to know exactly what your current installation

is or whatever type of computer you're using.

But this should give you the right information

to do the correct thing.

So let's go make sure that we have Rust up.

I believe it's just Rust up. Yep.

So we typed in Rust up,

and we can do rustup--version

and it'll tell us that I installed Rustup 1.27.1,

we're all good to go.

And we can see my active Rust version is 1.790.

So that is everything that you can see,

that like oh I have properly installed Rust on my computer.

And the next step that we need to do

is we need to install the actual Solana tool chain.

This is so that we can do a bunch of different Solana

related things like creating key pairs locally,

or compiling our smart contracts and more.

So we click here and it'll take us to the page

to do the installation.

So we can see here that there is a CLIcommand to install.

We just copy and paste it into our terminal.

So let me paste it right here,

and this will pull down the Solana binary

in order to run the Solana tool chain locally.

All right, so now we see that it was installed.

We've installed Solana version 1.18.18,

and it's telling us to do an export onto the path.

So we can just run this thing right here

and it will export our binary using Solana

onto our path variable.

This just basically allows us to use the Solana CLI.

So then we can open a new terminal,

let me make it bigger for y'all.

And we can type in Solana--version

and it should give us the version number.

And see here we have Solana CLI 1.18.18 installed.

So we're good to go.

We can also check in this other terminal just to make sure

that we did everything correctly on the Rust,

whoops version. There we go.

And we have 1.27.1 so let's clear this

and close out of this terminal.

All right, so we're back to here,

and we keep following the Docs pages.

The next thing that it has us do is we're installing Yarn.

You can also install NPM, you can install Pnpm, you can use Bun, you can do whatever flavor

of package manager you use for JavaScript. Doesn't matter.

This one says install Yarn.

I'm not gonna show this

'cause there's pretty good documentation out there

'cause it's outside of our ecosystem.

So go ahead and install your favorite package manager

for using no JS or something or JavaScript.

The next thing we need to do is we need to use

the Anchor version manager.

This is so that we can have the correct version

of the Anchor framework locally

in order to compile smart contracts on Solana.

So this just tells us to do cargo installed.

So let's go ahead and do that.

All right, and that's Anchor installed,

so you can see that it installed 030.1

and we can go open a new terminal

and see if it actually worked correctly.

So let's make this bigger for y'all.

AVM--version and we can see 030.1.

And in order to set your current Anchor version locally, 'cause this is just the version manager,

you do Anchor or avm use,

we're gonna use 0.30.1 throughout this boot camp.

And now we're using 0.30.1.

So that should have everything

that you need to get started, I believe.

Let's just double check with the docs.

All right, so this has some more information

to make sure that you have everything right.

It also tells you to do Anchor--version

to make sure we're using 0.30.1.

And so we have anchor-cli.030.1.

So now the next step is what we're gonna do

is we're gonna set up our project

for creating the voting app.

I'm gonna just make sure I'm in a good directory

that I like.

I'm gonna make a temporary directory for us.

There we go. Let's make sure.

Bootcamp temporary.

(keyboard clattering)

All right, so here what I'm gonna do is I'm going to do M...

Let's make sure it's bigger.

All right. So here what I'm gonna do is I'm gonna do npx

create-solana-dapp and you just type in that and hit enter.

And what this does is it'll create a scaffold

for building all your different projects locally on Solana.

So here we've first gotta enter our project name,

we're doing a voting dApp,

we're gonna use nextjs for this, we're gonna use Tailwind,

we might as well have the counter program. That's fine.

And it'll start creating the workspace.

And so once this finishes,

we'll have a template or a scaffold

in order to start building our voting app locally.

This is makes things a lot faster locally as well.

All right, so we've created our template Solana project,

and we can actually start running it.

So if you go, we can do LS here

and we can see there's a voting dApp folder now.

We'll go CD into it and if we do an npm i,

I used npm locally, that's fine.

So we can install the npm modules

to get our web front end running,

and then we'll do a npm run dev to run our actual website.

So npm run dev.

And it will open on localhost 3000.

So let's go look at localhost 3000,

see what it presented us.

All right, so this is basically a placeholder or a scaffold

for us to create our application.

This is what we're gonna use completely

to build out our voting application within Solana.

We have everything here.

There's a bunch of placeholder stuff

that we'll remove later on.

We're gonna focus on the program side first,

or the smart contract side and build that out.

All right, so let's go look at our other local stuff.

So going back to here, we're gonna close,

we're gonna open another new terminal.

So another thing in part of local

is that instead of running things on Devnet

or environment that's not on your local computer, we're actually gonna run a validator locally

that has a lot of the same capabilities

that you would see in a validator

on the actual live network.

And we can just simply do this with the Solana tool chain

by running Solana test validator.

So you wanna run this locally also to test

to make sure everything installed correctly.

Just as a note, if you're using Windows,

you will want to have this all installed on WSL,

otherwise it might not work.

And what you can see here,

the JSON validator is running on port 8899,

and it's processing slots.

You can see it's at slot.

What is this 1,300,000, and the confirmed slot

is moving forward and we're all good to go and it's running.

And so actually if I run this and I click here

and click on local, you can do things on local.

So you can probably, let's say we can see in accounts,

I can see that I have an account

with this much SOL on local,

no token accounts and everything.

This is the majority of your time

that's gonna be spent using the local validator

to build Solana smart contracts.

Another thing is locally you can use the Explorer

to look at everything that's happening

on your local validator.

You don't have to go do some API calls or something

to figure out what's happening.

You can all do it via the UI on the website.

And how you do that is you can go to explorer.solana.com,

you click here, click custom RPC URL

and that should be set by default

to your current local host, your current local validator.

So we can just click that and boom, we have that

and we can see a bunch of live stats.

So 1,300,915 and 16, we can see that roughly matches up.

This will be basically off of the finalized slot.

They're equal from the front end to the back end.

And using this Explorer,

you can see a lot of the different things that you're doing locally and making sure

that you're validating your transactions,

you're making sure that the instructions

that you're setting are correct.

This is the full local setup,

so that you can actually run all of your code locally,

iterate a lot faster

and not have to worry about other environments.

It's very important that everybody,

and this is what we're gonna be doing

throughout the whole bootcamp is everybody uses local,

the local test fighter and gets very familiar

with how to use it correctly, right?

So we have it running,

we have our developer environment running,

we have the Voting dApp running here on a website.

We can see I'm connected to it, we're all good to go,

everything is ready for us to get started on local.

This is all the setup that you needed

to start building local development on Solana.

What I like to do is before we build anything

on the voting dApp, I want to make sure

that we understand exactly what we're going to build.

Basically, we're going to architect all the different things

that we're gonna build before we actually write the code

just so that we kind of understand what we're thinking

when we build out the application,

so that we don't have to keep rewriting

the same code over and over and again.

So I'm going to draw it up here,

and we're going to give the full architecture.

Alright, so this is basically a whiteboard.

I'm gonna use this to draw out the voting dApp.

We're gonna create a voting application

that's gonna do two things.

We're gonna either be able to vote for an account,

we're gonna vote for crunchy

or smooth peanut butter basically.

So if anybody wants to vote

for their favorite peanut butter,

say you enjoy crunchy better than smooth, vote for that one.

And then if you wanna do smooth versus crunchy,

vote for that one.

It's a very easy way to showcase.

But you can kind of see like how this extends

to real life applications.

Say like an actual election,

or your school representative election or something.

You can actually decentralize this

to where every single person has a public

and private key like you learned early in this bootcamp.

And that is their identifier and you know that they voted,

and you can immediately count the votes that they have

for a specific candidate online.

So let's draw this up.

Okay, so part of this diagram or this architecture

is we have to understand what accounts that we need

for our smart contract before we write it.

So here, we'll be creating basically polls, right?

So we're gonna have a poll account, we know that.

And then we're gonna need candidates.

So we're gonna create another candidate account.

We're gonna have...

I think what we're gonna do is for each candidate

we'll have an individual candidate account.

So there could be multiple of these.

So let's just create a bunch of them,

so that it's easy to remember

and these candidates will have some information.

So each of these accounts,

since these are both different types of accounts

will have different information about what is the poll

and what is the candidate.

So the candidate that makes sense to have,

candidate name as who you're voting for.

And then maybe there is a candidate vote amount.

This is so that you can keep track on chain

via just looking up the account,

how many votes a specific candidate has.

Then the poll, we're gonna have a different amount

of data stored in it.

So we'll have like a poll name.

This can be like the question or the name

or yeah, this can be, let's call it poll question.

That'll work better.

Or we'll just call it poll. That'll work.

So we have the poll, like what are the question,

what's the answer or the question

that they're asking in the poll.

Like do you vote for, in our case,

crunchy or smooth peanut butter?

We'll have a description

that kind of tells you a little bit more information

about the poll so you can better vote

what candidate you want.

We're gonna have a poll start time.

So let's do poll start and poll end time.

And then we're gonna do one last thing

that helps us understand more information

about the poll and also do lookups later on.

So 'cause one of the things that we're gonna have to do

is we're gonna have to do lookups

on how many candidates there are within the poll.

So we might have five candidates,

we might have two candidates.

In our case we'll only have two,

but say we have hundreds or something, we wanna make sure

that we understand how to do the lookups correctly and very quickly.

We'll call it candidate index or candidate amount,

we'll call it candidate amount.

All right, so we have our two accounts and if you remember,

as part of the PDAs or Program Derive Addresses

that you used in the previous project,

we want to figure out how to find these accounts

and do lookups on them very easily.

So we're gonna have to derive these things differently,

so that it's very easy for us to do those lookups.

So let's get into that.

The way that we derive PDAs is we provide seats.

So in this case for the poll, we will have a poll ID.

This is so that we can get the poll,

and figure out which ID this poll is.

So this is like if you have a list of different polls

that you are doing lookups on, this helps you understand

which poll ID represents whatever poll

that you're pulling from.

So there's two ways that we'll have to do this later on

in the actual application.

You can either do just a incremental ID

or it's just like 1, 2, 3, 4 or five, et cetera.

Or you can do UUID as the poll id.

I think we'll just do 1, 2, 3, whatever.

And the way that we'll have to do lookups later on

is we can either cache it server side on the amount of IDs

and what are the polls and descriptions,

and then just do verify on chain.

Or we can just do like incremental lookups and figure out,

okay, what are the first 10 polls?

Showcase them on the website.

There's a lot of different things you can do with this.

We'll do a lot of that later on in the bootcamp

when we productionize this application.

But for now, we're just gonna pull ID,

it's gonna be a number and that's it.

And that's how we're going to derive the poll.

So this is the seeds for the poll account.

The other one is we have candidates.

And what I would like to do for this one is for the seeds,

we would always want to make sure

that it's identifiable based off the poll ID.

So that whatever poll that we are pulling from

with all the information, it's uniquely tied to that poll,

this poll ID directly ties it to that poll.

So we'll do poll id

and then just so that we can do lookups

based off of which candidate,

we'll do the candidate name as well.

So this ends up being our two seeds that we use

to derive each candidate.

So we have the seeds of poll id,

and seeds of poll id plus candidate name.

The whole reason and you have to make sure

that you give these different accounts a lot of thought.

But the whole reason why we're doing this

is so that we don't have to redo our accounts later

and redo the way that we do lookups later

because it's very important that we have good lookups

at the beginning of building around our program,

or our smart contract

so that we don't have any issues long term.

All right, so here we go.

We have our poll id and our candidate name,

and we have to also build out what are the instructions,

what are the functions we're gonna call

within the smart contract.

So let's go outline those out.

So the first one that we'll need is...

Let's pull this over here.

I'm gonna move some things around so it's easier

for us to view everything.

The first thing that we'll need

is we'll need an instruction.

So we'll call these instructions.

We'll need the instruction for initializing the poll, right?

So this will build the poll,

take out all the parameters in order to build that pole,

as well as the poll id, and it'll have everything we need

to build that poll.

So the next one that we need is we're gonna have

to initialize individually candidates on this poll.

So this is initialized candidate, right?

So we have the poll, we have each candidate.

The last thing, and the most important thing is you can't have a voting application without the ability

to vote for application.

So we have everything we need.

We have the instructions, we have each account,

now we can actually just start building.

So let's go over to our local environment.

You can open your favorite ID

and you can start building there.

So I'm gonna open Visual Studio Code,

and I'm gonna go open the specific folder

that I created earlier with my voting dApp.

Okay, so I have the whole code base open now

that we created earlier via the Create Solana dApp.

We can see just as a quick walkthrough

of this actual scaffold, we can see here, we have a web.

This has just a basic nextjs app, if you're familiar

with it or if you're unfamiliar with it, that's okay.

We will show you what's practical to get started.

You don't have to go end to end.

Basically, it just has a bunch of components

and within these components you'll be able to see.

So it was originally a counter page.

This is the page for that.

And then we have a bunch of information

in order to get the counter, we have a data access,

we have a feature, which helps us actually showcase

what that counter create is.

And then we have that UI for the page.

Don't worry about it if you don't know this too much,

we'll just go through what's practically needed

to build out the website.

If you wanna learn how to build websites or build the UI

or the front end for any of these applications we build

throughout this bootcamp,

there's plenty of resources online that you can use.

We're more focused on building the Solana side of things

rather than the UI throughout this bootcamp.

So we'll just create, we'll teach what's practical

throughout this, right?

So this is everything that's in the web folder.

And then what we're gonna focus here

is we're gonna focus on the Anchor folder here.

So if we go here to programs source lib, we will see that there's a counter program already based in here.

We wanna update that to our Voting dApp.

So we're gonna update this to voting.

I'm gonna just remove all of these instructions,

and then the rest of the accounts.

So you end up having this very small file.

There's barely anything in this file.

I'm gonna do a couple other things where we're gonna change.

We're gonna change this one to say voting as well,

this file, but even here, it also says voting.

Basically you can change everywhere

that says counter to voting.

If you use this in the future, should be automatic,

and then right now it's not.

So one thing to note is that we need

to also double check our anchor lang as the one

that we expect.

Throughout this whole bootcamp, we're using 030.1.

I don't know in the future

what the new versions of Anchor might hold.

So if you're ever in the future using this, say, I don't know, months in the future,

and we're going through this bootcamp

and you're using 031 or 032, and something doesn't work,

definitely make sure that you use the right dependency.

Also, as a general note, if you ever run into issues,

you can always double check the project link

in our GitHub repository.

It will be in the description below,

and you can go there and check all the code and make sure

that you're all set up correctly.

All right, so we have the right Anchor version,

we have our lib.rs.

The first thing I'm gonna do just to make sure

that I did everything correctly

is I'm gonna go cd into anchor

and I'm gonna do an anchor build.

This is how you build your smart contract locally,

and let's make this slightly bigger for people

to be able to view it.

And so this anchor build will compile our smart contract,

and I'm gonna do it early

just to make sure I didn't screw everything up.

I often will say build early and often.

You'll hear this throughout the whole bootcamp

on whoever's teaching so that you don't have to run

into issues later on and be missing a bunch of errors.

You'd rather run into the error as soon as it pops up,

rather than working backwards in your code.

All right, so this built everything.

You can see that it gave me a warning.

This is because I'm not using this yet.

We'll use it later, but that's okay.

We'll work through it.

So going back to our diagram,

we can see that we had two instructions

or three instructions actually, initialize poll,

initialize candidate and vote.

So we can just actually build those out right now.

So in order to build those out we can see that.

So I actually have Copilot installed.

I will also be using Copilot throughout this.

Highly recommend using it.

It's really useful.

You can already see that it's trying to get me

to create an initialized voting instruction.

I'm gonna ignore it and just do it myself.

But it's a cool thing to go through.

So we know that the public function,

and we had this as initialized poll,

we have to initialize the poll first

and we grab the context.

So this is context,

we're gonna call it InitializePoll.

We're gonna give it a poll id.

We're gonna make this like a U64.

So you can just have a bunch of polls,

and then we're going to return.

Right now, we're gonna return okay.

All right. So that's a base instruction.

I'm going to create underscores here.

So these underscores are basically just to denote.

I'm not using the variables just to remove the warnings.

We're gonna anchor build, make sure we do everything right

so we can see that something went wrong.

Let us make sure that we do it all correct.

Oh, it's because the initialized poll we haven't done yet.

So let's get started with that, right?

So one of the things

that I noticed actually before we get started

is that the program result's not correct,

it's actually result.

And that is correct.

So now if we do a clear,

so let's do a clear real quick and then rebuild.

It should just have the context there.

So there you go.

So you have the context of InitializedPoll

is not found, perfect.

So let's go build it.

So if you remember in the previous example

or previous project we built, we used the account macro

or the derive accounts macro.

So derive accounts.

And we're gonna create this as a struct

called initialized poll.

You can see it's...

Copilot kind of already knows what we're doing.

Doesn't know entirely,

but we're going to create it ourselves.

So struct InitializePoll,

we're gonna use a lifetime for this.

And then we have to create a bunch of different accounts

within this context.

As part of this initialized poll,

we're going to be creating accounts.

So that means we need two accounts already.

We need the signer and we need the system program.

Now, Anchor's really nice,

it'll actually pull in the system program already to you,

so you don't have to worry about that.

So we'll just need the signer.

So let's go build that signer.

So it'll be because we're getting money from it,

it's a mutable signer.

And then we just have this as a pub.

I'm gonna call it signer 'cause it's just nice.

Signer info.

So we know this is getting created, perfect, great.

We have a signer that's gonna be used to pay

for this account.

So next thing that we need to do, if you remember correctly,

we had the poll account.

So we need to create an account for the poll,

so let's go ahead and create it.

So account and I'll fill everything in just a moment.

We just wanna make sure that we set it up.

So poll, this will be account info poll, that works out.

And down here, I'm gonna create that struct real quick.

So I'm gonna use account, gonna do pub struct Poll.

And we're gonna put everything in this.

So we're gonna have a pub.

Poll ID, so it already kind of knows what we're doing.

Thank you Copilot.

So we're gonna have a poll id.

That's gonna be U64 as we mentioned earlier.

We have a description.

So this is pub description.

We're gonna call this a string.

We have a pole start, so pub pole start.

And that's also gonna be U64 'cause that's how you measure.

It's gonna be a Unix timestamp.

We have pole end, which is also gonna be U64.

And then we have a poll index.

We can have this or we had it as a candidate amount.

So candidate amount.

And we're gonna have this as U64.

So this is everything in our poll.

This should be everything that we need right here.

And so we should be able to create it right here.

Create the account right here.

So let's go build it all.

If you remember last time and Copilot

is giving us all the answers ahead of time

is that you can use init

to make a account automatically initialize.

The next thing is we're gonna have the payer.

We have to note which payer it is.

So we're gonna have to give it the payer.

We have to give it the space involved.

And so we had some variable stuff here.

We have a string, we have a bunch of U64s.

U64s, we know how many bytes there are,

but the string might be a different amount of bytes.

So we have to use something called max length to give

or it's in macro to give the max length

that this string could possibly be.

So I was gonna give that right here.

All right, so do that.

We are actually using a macro called max_len

and we can just give the number of characters

that this string can possibly be.

So I'm gonna just put 32 and that should be good.

I think that's good enough.

We could give, I guess we can do like a Twitter post,

280 characters.

We can put a whole bunch of description in there,

that should be good enough for this.

So we have the payer, we have the size

and we're gonna do another thing.

There's another thing called init space

that allows you to automatically calculate the space

and you don't have to do some extra work or math in your head to figure out

how much space this is called.

So in order to do that, you do hash,

it's gonna be derive InitSpace.

And so this basically allows you

to get the space very easily.

So when we do do the space here and do space, whoops,

it'll be always eight 'cause the way the Anchor does

in the current version

is that it always has eight bytes reserved to it.

So eight plus poll InitSpace.

I think it was actually INIT SPACE.

There we go.

And then we need the seeds.

So if you remember earlier, we have the seeds as a poll ID.

So we can actually pull that in and use that as a seed.

So we're gonna do seeds.

You can see as poll as ref.

It's not exactly what we're gonna do, but that's okay.

We'll update it, right?

So the way that you pull in different parameters

that you use is you'll actually use a macro called instruction.

And you can see polls in the poll id as a U64.

And then what you can do is you can then use the poll id.

I think it's as ref.

To do a seed for this poll account.

So this seeds sets up the poll id as seeds

to get that poll account.

And then finally, as explained earlier,

you always need the bump.

So we always have the bump in the poll,

we have the signer, we have the poll,

and I believe we can just run this now.

Let's go and do anchor build.

Build early and often.

Let's see if we run two errors.

We did run two errors. Let's go fix 'em real quick.

So traits, bump, blah blah, blah, poll.

Okay, so let's see.

Oh, let's keep going up.

Not found in scope.

So it looks like something is not set as the same thing.

Let's see, I did use that correctly.

And declared (indistinct).

We always have to go to like the top area.

Good thing about the Rust analyzer,

sorry, the Rust compiler,

it actually gives really good errors.

So it says here the program, okay, cool.

Initialized poll is not found in the scope.

So something is wrong with this initialized poll.

That's fine. We can get there.

All right. So we can see here, just looking at the error,

the top error is use program type

to add a system program field to your validation track.

This is because I'm creating an account,

I actually need the system program.

So pub system program,

it's gonna be a program of info system.

So let's go rebuild it.

Should have one other error and that is fine.

Okay, so now we have another error.

We can see that other error disappeared.

We can see that.

Oh, I understand.

I did payer, payer.

We actually called it the signer.

So it's just... That's okay.

There's some misspellings.

We do anchor build and looks like it's building.

So that wasn't that hard.

Just as a heads up, if you ever do run into errors,

definitely go to the Solana stack exchange.

If you can't figure it out

and ask a question about the errors.

It is very important when you do ask a question

however to provide all the code necessary

for Solana to reproduce your error.

If you don't give all the code, they might not be able

to have enough information to give you an answer

how to fix your code.

So right here we have anchor build,

and voila, we have initialize poll.

However, if you know it right immediately

you can say we don't do anything with the poll.

We have the context, we have the poll id,

but we just return okay.

Now, note with Anchor, and this took me a while

to figure out my first time

is that if you have this init, it does create the account.

So that is okay, however, there's nothing in that.

So we need to give some information to it.

So some of the other stuff

that we have is we have a poll start and end.

So we actually have to have that passed

in as part of the parameters.

So let's go ahead and add those real quick.

And I'm gonna take out the underscores

because we're gonna start using it,

using all these different things.

So next thing is we have a poll start.

This is gonna be the U64 we're gonna pass in.

And then we have a poll end that's also gonna be a U64.

And then finally we have this description

that we talked about.

And we're going to for this last variable,

this candidate amount,

we're going to use that to increment later.

We don't have to do that in the initialization,

but we do need to make sure that it's set to zero.

So let's go ahead and set it.

So we're gonna do create let poll

equals mutable context.accounts.poll.

This is so that we can update the poll account

and we're gonna set everything.

So poll id equals the ID we passed in,

the description is the description we passed in.

We will make sure the poll start is set

and the poll end is set.

And then the candidate amount, we're gonna set a zero.

So this is everything to initialize that poll.

So we have our first instruction, we're all good to go.

So an important part of local development

is testing your smart contract.

What we're gonna do is each individual instruction

that we build out on this voting application,

we're gonna test using a testing framework

called anchor-bankrun.

So let's go pull that up. All right,

so if you just look it up, it's called anchor-bankrun.

And you'll see that it is a testing framework

that allows you to test your smart contracts methods

or your instructions and make sure that you're expecting

the right values to come out.

So let's go test it out.

So the first thing that you need to do

is you'll have to do this yarn add anchor-bankrun thing.

I'm using a NPM.

So I'm just gonna do npm install,

use whatever your flavor package manager for node

is to install it.

So it is called anchor-bankrun.

All right, so we've installed it

and now we have to get started.

So if you go over here to...

Over here, there is a counter spec.ts, you go click on that

and there's already a test in there for the counter program.

We're not using a counter program anymore.

So we're just gonna remove all the information within it

and create a new test.

Let's go remove the whole thing

and remove some more of the code.

And I'm gonna call this or let's move the counter

'cause it doesn't exist.

I'm gonna call this voting,

and we're gonna call this initialize voting

or initialized poll because we're initializing the poll

with our instruction that we just built.

So going back to the actual code,

there's a few things that you need to do ahead of time

in order to make sure these tests run.

So one of the things is you'll have to go grab the program,

the smart contract binary

and paste it into a folder called fixtures,

so that it knows what smart contract to use.

We're gonna do this manually now,

however later we will do some things to make a trick

to where it automatically copies over whenever you do build.

So right here in target deploy you'll see a voting.so file.

We're gonna copy it,

and we're gonna paste it into a fixtures.

So create a new folder called fixtures and paste it there.

The next thing that I'm gonna have to do

is I'm gonna have to pull what's called an IDL,

which is kind of like an interface to help build

and generate language specific format

for calling smart contracts.

If you're familiar with like Ethereum or something,

it's similar to the ABI and those ecosystems.

So I'm gonna go grab that.

You can see here this is the voting IDL.

It is under target IDL, it's called voting.json.

We can see here in the example here

that it is const IDL equals require the puppet json

which puppet is the smart contract

that they're using in this example.

All right, so let's go copy it over.

So we're gonna do, what?

Const, yes, const IDL equals require

just like the other one,

except we're going to grab a voting,json.

So it grabbed it immediately.

The other thing that we're gonna have to do is we're gonna have to grab the program type.

So our program type is under the types folder,

which is under target, so let's go grab that.

So we'll just import it, import,

it'll be called voting in our case

from target types voting.

And so we have the IDL in voting,

now we can create what's called the context and provider

that allows us to interact with our smart contract.

So let's just copy these,

and we can get to using them in just a moment.

So I'm gonna put them right here.

You can see it's airing out, that's okay,

it's because it doesn't have startAnchor,

we can just import it real quick.

Say import.

Ah, you can grab it over here

and import at the top, perfect.

So it knows what it's talking about.

Our tests are local to our folder instead of this test,

which are in a specific test folder.

So we just leave this as nothing

because it's just gonna test what's there.

And then there's...

If you look here, there is extra program

which is added programs and accounts.

We're gonna not do anything for the added accounts,

but we'll do something for the extra programs.

We're gonna need the program id.

So I'm gonna create a public right here.

This is gonna be voting smart contract address

or voting address, let's call it that.

And we're gonna call it, it's a new PublicKey

And we're gonna copy the same public key

that we have within our code.

So if you go over to your lib.rs,

you can copy over the ELY,

for my case, it will probably look different

for your voting contract.

So now we have the address and we can put that in our thing.

And if you look at startAnchor added programs,

it has a name and a program id.

So we have to go add this right here.

So we're gonna do name, I'm gonna just call it voting.

And the program id is just gonna be our voting address.

There we go.

So we have the context, we have the provider,

now we have to create the program object.

So you can see here that the program object

is just created using the program type.

So our type is called voting with the IDL and the provider.

So this one's gonna be called voting program.

All right. So we have everything to get started

and Anchor makes it very easy to interact

with different smart contracts using the IDL

because with this voting program,

and you can already see Copilot telling us what to do

is we can actually just do:

await votingProgram.methods.initializePoll,

and then give our parameters.

All right, so we have initializePoll

and that's our method.

And if you look at our previous, our actual instruction,

it has poll id, description, poll start and poll end.

So let's fill that out.

So in order to do a U64 in TypeScript, you have to do new

and you can see it already knows what to do, anchor.BN.

And we're gonna have the ID of one.

The next type that we need is we need the description.

So we're gonna create this,

what is your favorite type of peanut butter?

The next two U64s that we're needing

is the poll start and poll end.

So these are actually Unix timestamps.

So let's just go pull a Unix timestamp real quick.

Unixtimestamp.com, copy the timestamp

and we're gonna do another new anchorBN, big number.

I'm gonna do this at the beginning of existence at zero.

And then we're gonna do another one.

We'll just add a bunch of time at the end.

So I'm gonna go grab...

Didn't grab this, so let's just grab that real quick.

There we go.

So I'm gonna add a bunch of time at the end.

This is just increasing numbers significantly.

This will be much in the future.

So if I were to like copy this and paste this to convert

to see what time it would be, this is in 2027.

So it's years ahead of our current time

and we're good to go.

So now we do the initializePoll.

The last thing you need to do

is you actually have to call the RPC with it.

So this .rpc after the methods call actually will call

the RPC to execute the instruction within the program.

Alright, so we've done that.

Let us test it and see if it all works.

So anchor test.

All right, so it tested, it ran, looks like it ran,

found test script.

Error configured port is already in use.

Ah, so one thing that we need to do is with Anchor tests,

we're already running both a validator,

a Solana test validator behind the scenes.

So we actually needed to skip it.

So skip local test validator I believe,

or maybe it's local valid.

It's always good to read the manual beforehand

to see if what you're doing is correct.

So you can see it's skip local validator.

And then we also need to do skip deploy

because we're using anchor-bankrun which allows us

to skip a bunch of steps and run directly

within the validator or what's called the bank part

of the validator instead of running on a very heavy lift

of the local test validator.

So let's all run that.

So anchor test skip local validator

and skip deploy.

And so this will run our test.

You can see here it's already, it's found our test script,

it's loaded our program, the one thing

that we'll see right here is that it's gonna fail

because it doesn't know what this voting program is.

And the reason being is you can see here it says counter

and that's not running

one of the recognizable program names.

So let's go update that.

If you go over to Anchor.toml,

you're gonna see that in here we have programs.localnet,

counter equals that address.

So we need to update that.

So let's go update that to our current program,

which we called voting.

Then we run the tests and if everything is done correctly,

it will pass successfully.

So we can see here it tested our voting program

from our binary file

and everything looked like it worked correctly.

It did the initializePoll check.

Now, note this is not a good test.

It doesn't do anything, it doesn't check anything.

So we actually need to go pull that account

and make sure that we did everything correct.

So one thing is we're gonna have to do

is we're gonna have to go grab that poll address.

So this is gonna be the poll address

from our program derived address because if you remember,

our seeds for the poll account is the poll id.

So we have to do this PublicKey.findProgramAddressSync, and we have to give the seeds.

So the seeds will be...

It'll be buffer.

Oh, actually it'll be new anchor.BN of one.

So that is the actual.toArrayLike Buffer

little Indian eight bytes.

This is how you grab the buffer from a U64.

So we have that.

And then finally we have to give the program address.

So this requires the seeds and the program address.

So that will be the voting address. Perfect.

So this should be a poll address that is gonna be created

based off of this initialized poll instruction,

and let's go verify it.

We're gonna first console log it,

we're gonna grab the poll account and then console log it.

So const poll equals await,

votingProgram account poll, fetch poll address.

So this is saying,

hey fetch the poll account off of this address.

And now we're gonna console log it.

All right, let's go see what it looks like.

All right, so we seeing it passed

and we can see right here we have a poll id of one.

We have the description there.

Poll start is the beginning of time,

poll end is this number.

And then candidate amount, there's no candidates yet.

So we have zero candidates.

So we've tested it, all looks great, it's fantastic.

Now we can do some expects,

so that this test is actually a good test.

'cause currently it doesn't actually test anything.

So let's do some of those tests.

So right now we can do two of them 'cause we have the poll.

We can go grab it

and you can see already it says poll id toNumber equal one.

And then we can do another poll.

It's not the question. Whoops.

It's gonna be the description equals this

what is your favorite type of peanut butter.

And then we can also do just for good measure,

we can expect that poll description,

pollStart to be less than, so let's go grab that.

ToBelessThan poll.pollEnd

because we know that it should always be under that amount.

So it's good.

Also, we have to change these numbers,

otherwise it won't work correctly.

PollStart toNumber.

Right, so let's test it.

Make sure that our test actually passes.

Right, so our test passed, you can see the poll got logged

and the poll account was actually initialized.

So we've done everything to initialize the poll,

now we can get to initializing candidates,

which will be very similar

to what we just did to initialize the poll.

So let's go back to the program, start building that out.

All right, so the next instruction,

if you remember from this diagram over here

is we have the initialized candidate.

So let's go build it out.

It's gonna be initialize candidate.

We have to create a new context

using different accounts.

Context InitializeCandidate is the context name.

We're gonna have the candidate name.

(keyboard clattering)

And do we need anything else for a candidate name?

We don't need candidate votes

because they're gonna be set at zero.

We also need the poll id,

otherwise we can't pull the right poll

that this is related to.

It's gonna return.

We're gonna just have a okay return first.

(keyboard clattering)

So let's just return okay.

All right, so we have that,

now we need to go create the actual account struct.

So that's once again we're gonna do derive accounts.

We're going to use, if you remember we're gonna...

One of the seeds is the candidate name,

and on the other seed is actually poll id.

So we need both the instruction information.

So this is gonna be instruction candidate name, poll id.

One thing to note at the current time you need to make sure

that they're in the same order

'cause it'll actually pull the parameters

after the context in the same order that they're elicit

in the actual instruction.

So if you don't have them in the same order,

you'll actually get the wrong data.

So just a quick note,

this will be pub struct InitializeCandidate.

All right, so there's a few things that we'll need.

So as we saw last time,

we absolutely need the system program

because we're gonna be creating a candidate account.

Same thing, we're gonna need a signer

because someone has gotta pay

for that new candidate account.

We're gonna need the poll because we have to use it

to increment the amount of candidates

that are within the poll.

And then finally we need the candidate account,

which we'll do in just a moment.

So one thing you might have noticed

is that in our poll account we originally had payers,

signer, SPACE.

Because we don't actually need to create it,

we're just referencing it,

we'll just do seeds equals the seeds and the bump.

Finally, let's go create our other account.

So this will be a new account, this will be a candidate.

You can see it already started pulling everything.

We'll have the poll id and the candidate name.

So it's just a separation or switch of these two.

And those are our two seeds.

Then we have the bump.

So then we go to pub candidate: Account info, Candidate,

and now we have to go create that actual candidate account.

So once again, account pub, struct candidate

and it's going to have candidate name and candidate votes.

All right, so we have the candidate,

we have all the information for the candidate account.

One thing that you know we used INIT SPACE again.

So we're gonna have to derive InitSpace

yet again like we did on poll on candidate.

And let's do an Anchor build just to make sure.

We've done a lot of code.

Build early and often, you can see I made a error here.

It's because I use string,

so we must use a max length.

We can do that again and grab this on the candidate name.

So I'm gonna say the candidate name,

the names are normally not longer than 32 characters.

So let's lower that to 32 characters.

Probably lowered even more to like 16

after a little bit of testing.

But that's okay.

You can see that we got a bunch of warnings.

These are just because we didn't use these different

parameters yet, but it did build successfully

which is a good thing.

That's why you always build early and often,

otherwise you have to go redo a bunch of code, right?

So we have a candidate name, we have the poll id.

Poll id is actually never going to be used.

So we make sure that it is underscore.

The candidate name will be used,

but it's gonna be used like this.

So we can actually make that underscore as well

to get rid of that error.

It's gonna be used in the instruction down below.

Final thing is we have the candidate.

So some of the things that we're going to have here,

actually the candidate name will be used.

All right, there we go.

So we're gonna set the candidate name in the account.

So let candidate equal

the mutable context accounts candidate.

So we grab the candidate account.

Candidate name equals candidate name

and then candidate votes equals zero

'cause you need to always start with zero.

Let's go build that, make sure it works all right.

All right, cool.

So it pulled this.

We have the initialize candidate.

All right, so now it's time

to test initializing the candidate.

So what we're gonna do

is we're gonna create a new test right up here.

It's gonna be integration test initialize candidate.

And we're gonna test making sure

that we're actually initializing

our crunching smooth peanut butter

and everything works as expected.

So we have this, I'm gonna also create

just for the future reference

'cause we're gonna have to vote,

I'm gonna create a vote test.

Because we know it's gonna happen,

so might as well have it there.

So we have a vote test

and now because we have multiple tests,

I wanna make sure that I pull this context provider

and voting program out so I never have to use them again.

So we're gonna quickly set them over here.

So this is context, let provider and let votingProgram.

And we're gonna do a beforeAll

and we're going to create a function that runs and sets these so that we always have them

before every integration test.

And we just pull these guys right back up there.

Oops. There we go.

Copy and paste, remove the const.

This is an async and we gotta remove these conts as well

because they're always gonna use the provider

and the votingProgram from above.

Actually we don't even need them at all,

so we just remove them. Perfect.

Alright, so let's run it,

make sure that it still runs.

So we don't wanna have to have done all that and it breaks.

That wouldn't be great.

All right, so we have all our pre-tests passing.

The initializePoll as expected

is doing what it needs to do.

Initialized candidate and vote don't do anything yet.

So I hope those passed. It did.

So now let's build the initialize candidate one.

So it should be very simple.

We should just be able to call

the initialize candidate instruction.

So if you remember the vote program

or the initialize candidate program,

it took a candidate name and then a poll id.

So we have to go put those both in there as parameters.

So this one's gonna be...

We're gonna switch it to be smooth first,

and then the number,

make sure you set the name first,

or set the same parameter order

that you had in your instruction

otherwise you will get an error.

And so this is our ID of our poll for smooth.

I'm gonna do this twice for both crunchy and smooth.

I'm gonna see if it runs, let's make sure it runs correctly.

All right, cool.

So it was able to create the crunchy,

and smooth candidates

and I believe if we run it again, okay,

so it's doing the initializeCandidate. Perfect.

Okay, so we've initialized those two candidates but that doesn't really mean anything

unless you actually grab the candidate.

So let's go grab them real quick.

So const, we need to go get the crunchy address.

It's gonna use findProgramAddressSync.

And if you looked at our seeds,

our poll id and candidate name,

so one of the seeds is already there.

The first seed is poll id.

Let's just verify with our lib.rs,

our actual smart contract.

The first seed is the poll id then it's candidate name.

So let's set that up correctly.

So poll id should be first, which is new anchor,

it's just comma.

Anchor.BN and it's what we had earlier.

So if we can just pull the same code

that we had earlier toArrayLike Buffer le

or the little Indian eight and crunchy.

So that's the crunchy address.

And then we can just...

Before we move forward, let's go grab that candidate.

Crunchy candidate equals await,

get the candidate account and let's go console log it.

All right, so let's test it.

All right, so you can see here the candidate name

is crunchy, candidate votes was zero.

Let's grab the other candidate.

So I'm just gonna copy and paste this code

and change everything to use smooth.

And then once we have all that set up,

it should be ready to go.

There we go.

So I've copy and paste it, now we have all smooth

and we should be able to see the console log

for both smooth and crunchy.

There they are.

They both have zero votes. Perfect.

So we have all those things,

now let's actually do some real tests.

So you have to...

I'm gonna set them all because we just initialized them.

The most important thing

is that the votes should equal zero.

So if you look over here, the candidate account,

let's go grab them real quick.

His candidate votes should be zero.

So it's gonna be candidateVotes equal to,

I guess this toNumber is equal to zero.

New BN, new anchor

'cause we have to make sure it's equal

to the U64 big number format.

Just make sure that works as expected. Perfect. Okay.

So expected double zero got zero.

So it is equal the same, we just have the wrong format.

So let's update that.

Candidate vote equal,

I think it's because it's already a beat number

so we have to make sure it equals the same.

There we go.

Let's see what it got.

Zero, zero received two. Interesting.

Let's see what we did wrong here.

So received expected, received was two.

Okay so it's probably because we did the...

Let's see, candidates votes, big number.

So I think we can just do zero here actually. Let's check.

There we go.

Because we send it to number...

There we go.

So we send it to number and it's equal to a number

versus the big number equivalent.

So this expect we can copy, and paste it right over here

and we can use it for the same smooth candidate.

And so now we have two tests to make sure

that their votes are zero

'cause that's the most important part,

that whenever you initialize...

They initialize the zero

'cause if they initialize it's more than zero,

that's basically cheating the system.

You're getting more votes for your candidate

than you previously had.

So next thing we're going to create a vote instruction,

so let's go create that right here.

So it's gonna be pub function, vote,

see it already knows what to vote.

This is candidate ID,

it's not actually a candidate ID.

We're gonna be voting for a specific candidate name

and there's our string.

It's gonna output a result.

One of the things

that we actually didn't do initialize candidate

now that I just remember is that the poll,

if you remember correctly, the poll had a candidate amount.

So we actually have to also increment the poll.

So let's go do that real quick.

Equals reference poll

and then do poll plus equals one.

All right, so that's set up.

Our vote amount is we have to grab the candidate

from the candidate accounts

'cause we're gonna set up this vote context

in just a moment.

We'll call it candidate,

so it's all the same and just plus equals one.

So let's go create that vote, vote context.

So once again derive accounts.

There is nothing.

Is there anything?

Okay, so we still need the candidate name,

it needs the poll address

because you have to vote for a poll address.

So let's grab that as well.

It's gonna be the same thing as over here actually.

There we go.

So candidate name, poll id.

So it's gonna be the exact same thing.

We're gonna have to pull a lot of the same accounts

except for you're not gonna have to create any accounts.

So we can remove the system program

'cause we're not creating accounts.

You can remove all of these initialized things.

So you just have the seeds and bump.

Do we need the...

We need the signer 'cause there's still a signer

on the address but it's not mutable

'cause it's not paying for an account.

And then we need the pub struct.

So we called this the vote context.

So we need to make sure that we set that correctly.

So pub struct Vote and let us anchor build it.

Let's make sure I didn't mess up anything.

It looks like I did run into an error,

so let's just double check.

Oh it's because I don't have the lifetime set up

on the struct correctly.

So that got rid of some things.

And then let's see, the public account could not resolve,

that's specifically the signer

because I'm not doing anything special with that.

Remove that and it should all work. There we go.

One thing that it said is that it didn't know what to do

with candidate name.

I think it's because it's in here,

and doesn't actually use anything.

We're just passing it in so that we can derive

the account successfully.

So now since we added the underscore,

it should compile with no error or no warnings.

There we go. It works.

So what we need to do, once again you copy and paste.

So copy that, voting that SOL over here,

putting that SOL over here into fixtures.

And then we should be able to write the spec,

the test for it.

So let's go vote for one of our candidates.

So what we can do

is we can do an await votingProgram.

What is it? .methods.vote.

And we have to vote for our specific candidate.

And we have to always remember to hit the rpc.

All right, so what we need to do is make sure

that we're always gonna pass the same thing.

So candidate name, then poll id.

So the first candidate name will be...

It'll be our same ones that we did earlier.

So right here I'm gonna vote for smooth

'cause that's the kind of peanut butter I like.

It's gonna have a poll id of one and that is fine.

And then we have the RPC, calls it.

And let's see if that worked.

Let's see if this part just worked.

Because that'll tell us whether or not

the RPC worked.

So let's do anchor test,

remember always to skip local validator

because we're using bankrun and then skip deploy

because we're once again, we're using bankrun.

So we get to skip a bunch of steps and see if it worked.

So it worked, however we don't know what actually happened.

We can see that the vote actually was working here.

So we know that it's actually voting, which is great.

We know that the vote instruction is being called.

We don't know actually what it's doing though.

So let's go grab that vote account,

and verify exactly what is happening there.

So we'll have to go and get these address.

So const, we'll have to go get the smooth address.

So it's actually...

We can copy and paste this up above,

and then just pull it from there and expect it equal to one.

So let's go grab it.

So smooth address, it is the same address as before.

Voting address. We have the smooth candidate

and we need to make sure that equals to one now

because we have voted to for smooth exactly one time.

So it ran to an error, let's see what it expected.

Expected received zero.

Okay, so let's figure out what happened here

to where it did not increment one.

Did we actually plus one? Candidate votes.

CandidateVotes toNumber, smooth candidate.

So this is fetching the smooth address.

It should have incremented because we did a vote.

Vote rpc. That should have worked.

Let's see expected.

Expected one received zero.

So something happened to where it did not increment.

So let's figure out what happened.

So we have the candidate, we have the candidate votes,

we did a plus equals to one and we sent an okay,

so why did this not increment?

Let's figure out.

So one thing that you can always do

is you can actually send messages or logs

to figure out how many votes something has

in your smart contract.

So we're gonna anchor build this,

and I'm gonna pull this over and verify

that this logic is working as expected.

If it works as expected, that's great.

We're good to go.

And we just gonna need to figure out

what else might have happened.

So let's go copy over the .SO again.

You copy and paste and let us test it,

and we should be able to get logs.

The program logs that it tells us exactly what happened.

So we go up here, we can see the votes for one.

So something was being pulled

that did not happen as expected.

So we got voted for one. That's great.

It came back. It was one vote.

We did get it afterwards.

I wonder if we...

Let's go pull the actual account address.

So smooth candidate fetch account,

address should have been happened after the vote.

It's perfect. Right.

So this is actually a really good point to make.

It did not increment earlier

because even though they error,

the library was saying I incremented

whenever we did this message.

And this is because all accounts

by default are immutable on Solana.

So unless you say that the account is mutable,

it will not actually update the account

even if you pull in the candidate account as mutable.

Doesn't matter because by default this candidate account

will also always be immutable.

So we actually have to go here,

and we have to make sure we declare the account as mutable.

And we didn't do this earlier,

so actually the poll account also didn't update.

So I'll add those two things.

So now that we added a mutable, so let's do an anchor build.

All right and then we gotta copy over our .so file again,

copy it over to our fixtures

and then let us run the test.

Perfect. So all three passed.

We can see that the candidate vote for smooth is one.

We can see it's not the errors,

but the logs went from zero to one here as expected.

And from here what we have

is we have a fully working voting smart contract.

We have tests that test it,

and that's all we're going to do for this project.

We have all the stuff on the smart contract side,

everything's interacting.

We've run tests on local,

we've run everything locally on the local test validator.

We have our full local environment set up for success.

And the next project we'll actually be using this vote smart

contract and putting kind of a front end on it,

but making it more social.

So look out to the next project,

and for that I will see you next time.

- For this next project, we're going to bring

the previous voting application

to people's favorite websites using Solana actions

and Blinks.

Solana actions are a specification for an API

that can return a transaction on Solana to be previewed,

sign and sent.

Whether it is a button, website or QR code,

actions make it simple for developers to integrate

the things you can do throughout the Solana ecosystem

right into your environment.

Blinks are blockchain links.

They turn any action into a shareable link,

allowing any website that supports links

to be the starting point of a Solana transaction.

As a result, blinks transform websites

or social media into a platform for executing transactions

all without taking the user elsewhere.

Any website could be the surface for your new application,

creating both a decentralized and user-friendly experience.

In practice, actions resemble a typical rest API call.

A user interacting with a website would trigger a get call

to retrieve information about what actions are available.

Following the list of available actions,

the user can then choose an action to execute,

which will then start the transaction signing process.

We will be making all our votes in our previous application

be actions allowing them to be executed

without a traditional front end with the help of blinks.

Now that we've had an overview of blinks and actions,

let's get started building them.

- All right, so with the project that we just completed,

the voting application,

we're gonna add something called blinks and actions.

There is a lot of different amount of work that you'd have

to do for blinks and actions.

We're gonna go pull up the docs right here,

just kind of go through all the ins and outs

of the technical know-how.

So first off we can see this...

It says actions and blinks as the title here,

we can see that to get started we need

to do NPM install@solana/actions.

So let's go and do that right here.

So we're gonna have to go outside of our Anchor folder.

I'll have to start with the voting dApp

and install our @Solanaactions.

And with what we're going to do,

we're actually going to need to add a couple steps

that we didn't do in the previous project to make sure

that the blinks and actions work for this project.

So one of those steps is we never deployed

our Anchor smart contract to our local test validator.

We were just using a Solana bankrun to do all the testing,

but now we actually have to test on a local test validator

to bootstrap some UI on top of it.

So let's go into the anchor folder,

and we're going to do anchor deploy.

Now this might be different based off your Solana config,

so actually you might need to check that first.

So if you do Solana config get,

you'll get a bunch of data based off

of what environment you are set up with.

If you wanna set your config environment to local, you would just do solana config set-ul

that will set everything to local,

so that you can just do anchor deploy.

So let's do anchor deploy.

And what this will do is it'll actually deploy

that voting binary that we use yesterday

to our local test validator.

And we can see here the program ID is that,

and let's go check it out in the explorer real quick.

So we go to explorer.solana.com,

we go set ourselves up on local and we go to our program.

We can see our smart contract has a program account

that addresses here.

We can see what the balance of SOL to store it was

and then when it was deployed,

and we can see a bunch of information

of like this is the upgrade authority of myself

who deployed it and a bunch of other information

of whether or not it's upgradable, it's executables

'cause it is a smart contract, et cetera.

And now that it is deployed,

we also will need to start setting up the actions

to interact with that smart contract

on our local test validator.

So we installed this actions,

this actions SDK, and the next step will be

that we have to set up this GET request

and this POST request to actually do something on chain.

If we don't have those, it won't actually do anything

on chain within these blinks.

So there's a lifecycle diagram here.

Basically what we'll have is that someone will do something

or click a button, it'll send a GET request

to get metadata on what are the available actions

that this user can do to your server.

This will show up on the UI.

The UI will then allow them to click a button

to perform an action, which will make a POST request

with their specific account public key

to the action provider or our action server.

We'll then respond with a serialized transaction,

we'll show the transaction details to the user,

allowing them to sign that transaction,

and then they can approve it if they want

and it will actually interact with the Solana blockchain.

So that's the full end to end

of how blinks and actions works.

So first off, let's create this GET request

to get the right information to do our actions.

So let's go pull down here.

We can see there's a bunch of information about blinks,

we'll get to this in a moment.

But ultimately our GET request has to have

a Solana UR Action URL that describes an interactive request

for the signable Solana transaction or message.

So let's go ahead and do that.

So what we can do is we can go into our web app, API

and create a new folder called, we'll call it vote.

I'm just gonna copy and paste this route,

so that we have a single GET request. That's fine.

And if we ran this successfully,

which if you're not already running your server,

you would just do npm run dev

and that will run both your front end,

and your server side.

We can go ahead and check

if this GET request actually exists

by going to our local host and going to, what is it?

API vote.

So let's go ahead and check it out.

So if you go to localhost3000 should be api/vote

and this should return some information.

Hello from the API.

This is the very basic nextjs server side work.

Just as a note, we are creating quite a heavy application

for these blinks.

nextjs server side is quite heavy for this usage.

Recommend like a node JS server if you can,

like a express server.

But because we have their scaffold

and we will later on in this course build

the actual front end for the voting application.

We're just gonna use this backend

and bootstrap it with this voting action.

All right, so we got some information

from our server using the API vote.

Next up, we need to actually return valuable information

in order to display the blink on any social media.

So let's go ahead and do that, right?

So if you go to the Solana documentation again for actions,

you can see that the metadata expected

is this action GET response.

Has an icon, a title description,

your label has some links and linked actions

and then an error.

So there's a bunch of different information

that you can respond via this GET request.

So if we go over here, we can go ahead and create that.

So we can do a const, what is it?

We can call this the action metadata equals of type.

This is actually type action GET response equals,

and now we're gonna start filling in our metadata

Action manager is required.

Missing properties, yep.

So it's just saying, hey, you're missing some properties.

Icon title, description and label.

So we're gonna fill those in real quick.

I'm gonna fill them with nothing

just so that everything will play nice.

And then we'll actually fill in with the real information.

So it's icon, title, description,

and there's one other thing and it's label.

All right, so we have some basic action

GET response metadata.

Now we actually need to fill it in.

So first off, what we're doing for this

is we're creating a voting application that can be viewed

and we need to create an icon

that is choosing between crunchy,

and smooth peanut butter.

For completeness sake

or being able to get it done really quickly,

we're just going to look up a peanut butter icon image

and pull it from the web.

So if we go to just a peanut butter image,

I can just go grab some peanut butter.

This one looks fine enough to me.

And I'm just gonna copy the address,

and we're gonna use that as the icon.

All right, so now we have the title.

We're gonna say vote

for your favorite type of peanut butter.

That's gonna be our full title.

And then the description is vote between crunchy

and smooth peanut butter.

All right, so now we have this label, we can add something.

This will be for a button.

We'll just put it as vote for now.

We're gonna have to do some extra work

in order to make it actually usable on the blink,

but I just wanna showcase what it looks like.

All right, so in order to return this response,

we're going to return Response.json,

and give back our action metadata.

So there's our response, everything's good.

Now, we have this GET response

and let's go test it real quick.

We can just refresh this GET request.

We can see here, hey, there's a nice little JSON

that it got returned.

It has an icon, a title, description, label.

Perfect. We have everything there.

Now let's actually see it in the blink.

So in order to do do that,

there's a website called dial.to.

And you would do to test locally,

the query parameter is action, solana action:

and then whatever your local host API is.

Ours is localhost3000 apivote,

so let's go look at it.

All right, so this is what's gonna load your actual blink,

so that you can view it and see what it might look like

on a social media like Twitter or Reddit or whatever.

Now you can see it's not loading.

Now I meant to run into this error

and you'll see that it says something about cores.

Look at that.

All right, so the vote ran into a cores error.

This is because, what is it?

Cross, cross origin resource sharing.

It's to help protect you.

We're going to do some things to get rid of this error, and this is just a common error people run into with blanks.

So let's go and add that.

What's really cool is we can just add this as headers

and it should be part of what is in the Solana actions SDK,

which provides a very nice headers for you

to already use, right?

So what you would do is you would have headers

and you would have actions course headers.

You can see it's coming from the Solana actions SDK.

And so that should now give you all the information

to properly serve your action via a blink.

All right, let's go look at it within our dial.to.

There we go.

So we can see right here is we have our nice little image

of the peanut butter.

We have the title:

vote for your favorite type of peanut butter.

And then the description:

vote between crunchy and smooth peanut butter.

You can see that this part says

the action has not been registered.

Part of the whole system for onboarding blinks today

onto social media is you have to actually get your action

registered via some kind of registry out there.

There's one that's managed by Dialect.

There might be more by the time this video is live,

basically it's not registered,

it's, hey, use at your own risk.

We don't know what this actually does.

That's okay for now.

So we have our blink, this is what it looks like.

Looks really great.

If we hit vote, nothing's gonna happen.

It's gonna say failed to fetch.

And that's because whenever you do the vote,

we can see here via the network tab, it's gonna try to,

well one, it's gonna do a course error,

and then it's also trying to do a post and it's failing.

So let's go and get through that.

So one thing to note is that as part of the full end to end

for getting the post, we're going to also have to create

an options return request.

And what we can do with that to make it very easily is we can just make it equal to the GET,

so that it knows everything that it needs to do.

So let's go ahead and do that real quick, right?

So in order to do that, you would do export const OPTIONS

and just make sure it equals the GET.

And so now when we go back over here,

and we refresh, let's go inspect.

It should just fail the POST now.

There you go.

So the POST failed, the method's not allowed.

It's because we haven't created the POST yet. Perfect.

Run into all the errors that we expected.

Now we actually have to create the POST.

But before that,

you notice that our GET request just has vote.

This is actually not voting between crunchy

or smooth peanut butter.

It's just vote.

That's not enough information to present on your blink

for someone to make a good enough action.

So what we're gonna have to do is,

if you look in the actual documentations

is there's something called links

and there's these linked actions

that basically give a list of related actions

for this user to possibly perform.

So let's go and add that real quick.

So we're gonna do linked actions and it's a a list.

So links, it's gonna be a list of possible actions.

So let's create a actions list as well.

This is gonna be linked actions. Whoops.

And let's go and look at that

and see what it's supposed to look like.

Just as an example, it's always good to figure out

what these things are supposed to look like ahead of time.

So you can see here the interface

for a linked action has the endpoint

for the action, has a label.

This is different from the original label we had

and a parameter for acceptable input if you want

to give input on the actual blink.

So we're gonna have the action,

and we can see it's a list of linked actions.

We can see here it's already bringing some things up.

We can pull these out

because it's only, let's say it's only href and label.

And we have this vote crunchy and vote smooth.

So we're gonna do vote candidate equals crunchy

as our query parameter

'cause we're not gonna create different posts

for each of them and vote candidate equals smooth,

close the bracket and we should have everything else.

Not find a name action. That's okay.

Let's figure out what's going on.

Property of missing action,

require type linked actions.

So let's go and update this,

so that the compiler's happy and that'll just be fine.

So the way that it's actually supposed to be

is it's just because I had a bracket instead of curly brace.

So now we have the total actions that some user can perform.

We have the label, the original label of vote,

it's not gonna be used anymore.

And then we have the links

and actions of what someone can actually do.

So let's go ahead and do that real quick.

Let's double check.

Yep, we can see all the extra information

and let's go look at our blink, see if it's updated.

And you can see right here we have a vote for crunchy button

and you vote for smooth button.

So now our blink looks as expected

and we have everything set up on the metadata

in order to execute our blink correctly.

Now, as you probably can guess,

still will fail with a 405 on the POST

because we have yet to create those posts.

So let's go and get started on that.

So we're gonna create one single function

export async function POST.

And we're gonna get the information from the request

from specifically the URL parameter so we can understand

whether or not the candidate is going to be a crunchy

or smooth peanut butter.

So you can see there's already a...

Copilot is already trying to give me some information.

I'm gonna ignore it because I wanna write my own.

All right, so the way that we do this is we would first grab

the URL and this would be grabbed

from the request that came back.

Next we're gonna grab the search parameter,

which we had as candidate.

As we saw up here, the candidate equals crunchy.

And now we need to make sure that the candidate given

is actually a valid candidate.

So let's make sure that the vote

is a either crunchy or smooth.

That way we have a good return back to the user

and if they somehow enter something wrong,

it's not gonna give 'em a weird error.

We don't wanna surface strange errors

that don't make sense to the user, right?

So we're gonna do if a candidate is not equal to crunchy

or smooth, we wanna say invalid candidate.

A return status of 400 and editors

will have to also once again be those action course headers,

otherwise it won't work.

All right, so we've done all of our testing

or our validation of input, we now have our candidate,

now we have to actually create the transaction to send back.

So in order to create transactions,

we're gonna have to need a few things.

We're gonna need a blockhash,

we're gonna need the message

and we're gonna need any signatures

that we possibly need to send to be added

to that transaction.

Thankfully this transaction does not need any signatures

on the server side.

Sometimes maybe you wanna sign something server side.

In this case we don't care.

So I'm gonna create a connection.

This will be part of Web3.js

and we're gonna have a connection off of http

is gonna have to be 127.0.01 at 8899.

This is our local test validator

and we're gonna use the confirmation status of confirmed.

And so what this, sorry, it's a commitment status.

So commitment status,

that means is we have a few commitment status levels.

There's a process level, there is a confirmed level,

there's a finalized level.

So those levels are telling you how certain they are

of a transaction making it on the cluster.

The process ones, hey, I found it, it's not yes, confirmed.

Confirmed means it's confirmed by 66% of the actual cluster,

and then finalized it's confirmed by 66%

and there's been 31 blocks afterwards

that were also confirmed.

So finally this takes a lot longer.

Confirmed is generally safe.

At the time of recording,

there's never been something that was confirmed

that didn't make it to finalized.

All right, so next up what we're gonna have to do

is we have to grab all the information that we need to

from the URL, namely the account

that the user signed with on the blink

in order to create the transaction successfully.

And so the body of the POST request,

if you go look at the POST request

that is being built over here,

let's go look at the body of the POST request.

Payload. You can see that it is giving my public key

that I have tied to this wallet to be used.

So we have to go grab that right here.

So it's gonna be body of type actionPOSTrequest

equals request.json.

And I believe it's gonna be...

This is a promise, yeah.

So it returns a promise, so we need to await it.

And so now we have the body,

now we need to go grab the account.

So const voter equals body.account.

So this will have a type of account on the body.

Now we wanna make sure that this account

is an actual valid account.

So one thing that we can do real quick

is we can do new PublicKey

'cause we know it should be a public key

on the body of account.

If it's not a public key, this will error and that's fine, that's to be expected.

So cannot find quick fix just imported

on Web3,js if you haven't already.

Now if this does air, we wanna make sure

that it airs gracefully.

So let's go ahead and add it over here.

So I'm gonna do...

We're gonna do a try catch right up here,

and I wanna make sure that the voter

is actually a public key and give back an error.

That makes sense.

So we're gonna return right over here,

or we're gonna try this, make voter equals

new PublicKey body account just like we did up above.

We'll move it right over here.

So if that passes, we can just move on.

If it doesn't,

we wanna make sure we return a valid response.

So it's gonna be invalid count 400, battery request,

same once again with the CORS HEADERS.

All right, so we have that error. Perfect.

Now let's actually start building that transaction.

Now if you remember from testing,

and I can just pull this up real quick.

We used basically a vote program

in order to initialize the candidate.

We did some information on, let's see,

on voting for the candidate.

What we're going to need to do

is we need to grab specifically this method right here.

And we need to pull the instruction.

This right here, right now just sends it

out as an RPC call. We don't wanna do that.

We wanna actually use it as an instruction.

We also need to make sure

that we are adding the signer correctly,

and it equals the account versus us signing

on the server side, 'cause in our case we don't care

about signing on the server side.

So we have to do exactly what we did

with some additional differences to create

that votingProgram, right?

So like here where we are creating a provider or grabbing the IDL, the voting type,

we're gonna have to do the exact same thing over here.

So let's go ahead and pull that out.

So we're gonna pull out IDL equals your voting JSON.

I'm gonna make sure that, that's actually where it is.

I don't think that's...

I don't think Copilot gave me the right answer.

So this is going to be backwards, backwards, backwards.

I think one more actually.

Yeah, backwards anchor, target, idl voting.json.

So you can think of that like this is a lot.

What's great is about on nextjs, you can just do this at

and we'll pull you back to the web folder,

then you can just go back and move into the Anchor.

So we have the IDL, perfect.

Now let's go also import voting.

The voting type.

And it's from target types voting just like we did

in the test.

So now we have the voting type, we have a IDL.

And now this is where it kind of difference....

It's a different thing because we need to create a provider

to create this program.

And we don't actually care

to have a bankrun provider 'cause that's used for tests.

We need to make sure that we use the right Anchor provider

in order to create our program

on our specific POST request, right?

So up here what we're gonna have to do

is create the program.

So program of anchor.ProgramVoting,

'cause it's our voting program equals new Program.

Don't actually think we need to do this anchor thing

'cause we already have it imported, which is nice.

Let's pull it in.

New program of IDL and it's asking for a provider.

Our providers just gonna be our connection.

So we need to make sure

that this is after when we create the connection. All right?

So let's go look at what they are we got.

Looks like it doesn't really know what this is.

So let me go.

Expression is not constructable,

no construct signatures.

So this is probably something to do with nextjs.

So part of nextjs, what you're gonna have to do

is you actually have to go update the nextjs config

in order to properly do it.

Also, I have two news here.

Oh it just worked out of the box.

So normally in case you run into this error,

'cause sometimes you will, you're gonna have to update

the nextjs config in order

to properly compile this correctly.

This one we didn't have to

because it seems to have recognized it perfectly.

It's great. But just as a note,

if you're running into problems with this specific one,

you can just look it up.

It's a common error.

You'll just copy and paste your error

and you will be able to find it very quickly.

But we have the program.

Now we need to grab the instruction for voting.

So we're gonna grab the instruction right here

and it's gonna be program.methods.vote.

And if we remember from our counter spec,

our test earlier, we're gonna need the...

To vote, we're gonna need our specific ID

and the actual smooth or crunchy.

So let's just double check smooth and crunchy.

Where you wanna make sure

that it says crunchy capital and smooth capital.

Otherwise it won't be pulled in correctly

via our smart contract.

So we have crunchy and smooth.

So we're gonna go put that in there.

That's gonna be the vote...

It's gonna be the candidate if we set it correctly.

Yep. Candidate.

So we're gonna vote the candidate

and then new Anchor, oops, .BN of one

because that's gonna be the same thing as we had earlier.

Anchor BN of one so that we have the right information

of the poll id to give.

So it's saying that I haven't imported it, so let's go import anchor real quick.

Actually we can probably just pull in BN,

let's just pull in BN from Anchor.

That would be even better. There we go.

So we have the vote.

The last thing is we need the actual instruction

'cause we're not gonna be calling the RPC.

All right, so we have instruction.

This returns a promise of instruction.

Let's just await it

because we don't want to just have a promise.

So there we go.

We have our vote instruction with the given candidate.

Now the one thing that we don't have on here

is we don't have the accounts.

So one of the accounts that is required

is we're gonna require the signer.

And the signer is gonna be the voter.

And so that's all we need to add is the signer account.

This is so that the instruction knows

that hey, I'm not signing it here,

you're gonna be signed somewhere else.

And we should validate that the signer equals the person

that passed in the original request via the blink.

Alright, so we have the instruction.

Now we need to get a blockhash.

So let's go in Const blockhash equals,

it's gonna be await get recent,

not get recent blockhash,

but it's gonna be a different RPC call.

Let's go pull that out real quick.

So not get recent blockhash, but get latest blockhash.

And so this will give us some context.

You can go look at this return,

basically it tells you whenever this blockhash expires.

If we haven't learned already,

blockhash is expired after 150 confirmed blocks.

So if you're trying to use it and it's too late, it's gone.

This is to prevent replay attacks on Solana.

It's all good to go.

All right, so now let's create our transaction.

So const transaction equals new Transaction.add instruction.

And so this is a new type of...

A new transaction.

It's gonna say that I need it from what they address.

There we go.

We have the instruction,

now we're missing some things on this though.

We have to be able to say, hey, what am I expecting

and what's my blockhash?

So let's go ahead and add that as well.

So the way that you do that is in here,

you're gonna add some information.

You can see that it already knew some of it,

it's gonna be the blockhash and the feePayer.

But you can see, oh this is not a valid thing

'cause it's a deprecated constructor.

The one that we're going to be using

is, I believe it's this one.

Let's go pull that.

So we need the feePayer, the blockhash

and the last valid block height.

So let's go and add that.

So we have blockhash equal to that blockhash and lastValidBlockhash, block height,

which will be blockhash

and there should be a blockhash lastValidBlockHeight

on the blockhash we got earlier.

There we go.

So we have the transaction now,

and finally we have to return it

as an actual response back to the user on the blink, right?

So let's go and do that.

We're gonna do const response

equals await createPostResponse.

This is a thing from actions.

This is going to be a different type.

So let's go ahead and pull that out and see what it expects.

So it expects your fields.

This field is just gonna be the transaction,

that's all we care about.

So this is gonna be of type fields.

Transaction equals our transaction.

All right, so we have our response.

This is the fields that we have in it, just the transaction

and finally we need to return it.

So return response.json, and you always remember

to add the course headers,

otherwise it's always gonna fail, right?

Let's go and test it out

and see if we did everything right the first time.

All right, so we'll go refresh this blink.

We can see here it has the vote for crunchy

and vote for smooth.

And if I click one of them,

it should pull up my wallet in order to sign

for the transaction.

There we go.

All right, so we're here signing for the transaction.

If you run locally,

there's gonna be a lot of information here.

What what you find usually with wallets,

so they're not doing any like caching on local,

they're just skipping it.

We can just confirm it

because we know that we're working on local.

Oftentimes if you do see these errors though, I would not confirm your transaction just as a general FYI.

So let's confirm it and see there's a...

So bunch of errors showed up, but that's okay.

The response got back as 200 okay.

And these errors are just...

The wall wasn't able to figure out what was going on.

And so we have now a end-to-end blink.

So let's go ahead and check that on our explorer.

So let's go pull up the transaction history,

and we should be able to see different transactions

on this program.

So looking at the program,

let's see if we can refresh a few times.

All right, so let's go ahead and figure out

what might've went wrong.

All right, so the reason why this did not work

is because what's happening is the...

We're doing this transaction, everything is correct here.

We have all the correct information,

however, we have a poll id of one.

Now if you remember, we just deployed our program earlier,

our smart contract onto our local test validator.

We don't actually have any information

about this actual smart contract,

or these poll or these candidates.

So we actually have to create those.

A very easy way to create those

is you can just run your test.

And if you write your test correctly, which we did earlier,

this should be able to create the correct candidates

of smooth and crunchy for people to vote on.

So let's go ahead and run that real quick.

And the way that you would do that is anchor test,

skip local test validator,

skip local validator.

We'll deploy, that's fine to me.

So it's gonna deploy to our local test validator,

and then it's gonna run all of our test.

And these should set up our specific poll onto our local.

So it looks like that they're not pulling,

oh, it's because it uses a different program id

than the one that we actually deployed.

So we can go pull that right here.

So you can see it's not ELY, it's actually 6RM.

So we just have to replace that and that's fine.

So let's do that again, anchor deploy.

And then we'll run the test suite,

and we can see...

Let's see what would failed this time.

DeclaredProgramIDMismatch.

So this is saying that something did not match correctly

to where the program that was provided did not equal.

So the reason why this vote for crunchy,

and vote for smooth button does not work

is because we never set up our poll id,

or our poll and candidates early on.

If you remember correctly,

we were using Solana bankrun in the previous project

to do a bunch of tests.

We then, early on in this project when we started,

we deployed our smart contract

onto our local test validator.

But we have yet to actually set up any of the configuration

in order to create our votes.

So let's go ahead and start doing that.

So I'm gonna do it via the test.

We're gonna change some things.

I'm gonna remove this code right here,

and we're going to use the test

to set up our configurations.

So we have the vote program.

This vote program is going to be equal to a new thing

using our anchor workspace.

So what we do here is we do it as anchor.workspaceVoting

as ProgramVoting.

So this creates our correct program.

And second thing is we're gonna have to make sure

that we're setting the provider correctly to run locally.

So set provider,

this is gonna be a specific different provider

'cause we're doing it locally.

So this provider is gonna be anchor.anchorProvider.env.

All right, so now what we can do is and sometimes

if you don't do this, you'll have to do an anchor clean.

This will clean up your workspace,

make sure everything is set up for success.

I'm going to go ahead and do that as well.

And then we have to run it

to where it's building the program, deploying it,

running the test, and creating our configuration.

So let's go ahead and do that.

So once that completes,

you should have a bunch of test pass and the test,

you can see that hey, we have all these crunchy

and smooth peanut butter set up.

And you can see here

that there is a...

All the program ran their instructions correctly.

If you do run into error, make sure this public key

is equal to the deployed program.

You can always check what program public key

that you are set up, but make sure it is always equal,

otherwise you're gonna run into issues.

So let's go ahead and test our blink real quick.

All right, so let us vote for smooth.

It'll pull up here and it will send the transaction over

to our account.

Let's go over to our program account,

and we should be able to see if we did everything correctly.

A new transaction, that at the very end it's voted

for candidate smooth.

Currently I have three votes on my local one for smooth,

and everything was created successfully.

So that is how you use Blinks and Actions.

A very quick tutorial

of how you create Blinks and Actions

with a application on chain.

To recap what we did is we took over our voting app

from the last project, we extracted a lot of the code

of how to interact with the program.

We created a server side component via nextjs

to serve the actions and allow people

to get the serialized transactions.

And then finally, we were able to view

the Blink via Dialect Blink's tester website and interact

with it to test everything out.

Everything seemed to have worked correctly,

we were able to vote for our different peanut butters

based off of what we liked

and it was all done via this blink.

Now, if you wanted to take this

to production this specific blink,

you would have to go through the registry and register it.

And also I'd recommend deploying this

to an actual environment, maybe Devnet or even mainnet.

And that way other people can interact

with it via whatever social network they please.

Yeah, that's everything for this project

and we will see you in the next project.

- Our next project will be a CRUD decentralized application.

CRUD stands for create, read, update, and delete,

which are the four basic operations of persistent storage.

This is foundational,

and we'll help you understand data handling,

smart contracts, front end integration

and blockchain transactions.

These skills are not only essential

for blockchain development, but also highly adaptable

to various real world applications.

These skills can be transferred

to any sector needing decentralized solutions like finance, healthcare, and more.

Understanding credit operations on Solana allows

for innovation as you can modify or extend your application

with more complex features in the future.

So let's get started.

Okay, so we're gonna get started

with using the MPX CRUD Solana dApp command

because we're going to be creating a backend connected

to a front end.

And this scaffold has your anchor program

with a basic counter program and it's already connected

to a front end for you using nextjs and Tailwind.

So let's start that.

We'll type into our terminal and npx create solana dApp,

and you'll be prompted to enter your project name.

So we'll just name this crud app.

Oh already exists, I've done this before.

So we'll name it crud app two.

So we'll select a preset, Next.js.

And we're gonna use Tailwind for our UI library.

And then here you can select an Anchor template.

So we'll do the anchor counter program with tests,

and now it's gonna be creating the workspace for you.

This is gonna take just a second to load,

so we'll just wait for it to load.

Okay, so now you can see that your scaffold is loaded,

and you can actually see there's a command here

that says Found Anchor version 30.1, please upgrade.

So this scaffold currently is set to anchor 30.0.

So we just wanna make sure

that we're on the latest version of Anchor.

So let's open up the scaffold.

And here is your Anchor folder.

And the Anchor folder

is going to have all of your backend smart contract code.

And for the front end you'll have the web folder.

So we'll touch the web folder later on.

We're gonna focus on Anchor right now.

As we just talked about,

we need to update our Anchor version.

So we'll go into the Cargo.toml

and update to the latest Anchor.

So here you see Anchor link is set to 30.0.

We'll just update that to 30.1.

And now looking through the Anchor directory,

if you go into programs, counter, source, lib.rs,

that is where your smart contract lives.

And what we're gonna do is just update

the counter to our CRUD app.

So we're gonna rename this, to be a CRUD app.

And then in the lib.rs,

with this preset you have a working counter.

So you have closed, decrement, increment, initialize

and set instructions.

Since we're updating this to a CRUD app, we don't need it.

So we're just gonna delete all of our instructions here

and just have the basic layout

for a smart contract using Anchor.

And if you see all of these data structures underneath,

they're all related to the counter program,

so we can delete that as well.

And now we have a basic layout.

We're using anchor link from our CRUD,

and then this declare ID has a generated public key.

This public key is going to be the program ID

of the Anchor program that you deploy on Chain.

So this will always be your program ID.

When you run Anchor build,

it's going to update accordingly.

Inside the program macro is where all of your instructions

are gonna live.

So that's where we'll write the CRUD app instructions.

But first, anytime I write a smart contract,

I always want to start with the program state and the state

is where you're gonna hold all of your data.

So as you know, smart contracts on Solana are stateless.

So all of the state is stored inside program accounts.

So we're going to create an account for the CRUD app

and define all of the data that we wanna store.

To do that, you can use the account macro with Anchor.

So we're gonna specify an account here,

and then we're going to define the data structure.

So I want this to be my JournalEntryState, okay?

Now we're doing a journal here

just because that's a very basic example of a CRUD app.

In a journal, you want to create journal entries, you wanna be able to update them if you need to

and be able to delete them if you need to.

So we're just using a journal as our example.

And let's see, what do we wanna save

to the state of a journal?

We wanna know the owner of the journal, so we can save that.

And an owner is gonna be of a type Pubkey

because it's connected to your wallet address.

When you're writing a journal,

usually title it and you have a message.

So let's save those as well.

And these will be of type string.

Okay. Now when you're creating a account on Chain

and you have storage, you have to pay to hold that storage

on Chain based on how much space it takes up.

So the rent is directly correlated to the space on Chain.

So we have to calculate

how much space our state data structure

is going to take up on Chain.

You could do this manually.

Each type has a different number

for how much space it takes up on Chain,

but Anchor actually does this for you.

So we can use the deriveInitSpace macro

to be able to calculate how much space is on Chain.

And you can see when we do this, we generate two errors

and the errors are for the string types.

This is because strings can be infinitely long.

So that's a little hard to calculate space there.

So we need to set a max length for each string value.

So for the title, that can be a little smaller.

I'm gonna set the max length of 50. Okay.

And then you can see the error goes away.

And I'll set a max length for a message as well.

And we'll say a thousand,

maybe you want a really long message.

Okay, and now that error went away.

So this is our state's defined.

This is everything that we'll need to store

for a journal entry.

And it's going to live inside our journal entry account.

We need to create this account, we need an instruction

to initialize a journal entry account.

So that's what we're gonna do next.

That will be our first instruction handler.

We're gonna do a pub fn create journal entry.

Whenever you're writing an instruction for Anchor,

the very first parameter that needs to pass through

the instruction is going to be your context.

The context macro is used to define a struct

that encapsulates all the accounts that'll be passed

through a given instruction handler.

So we're gonna write a custom struct for this instruction.

So we'll write the context,

and then we'll name the struct to create entry.

Okay, just setting up the instruction here.

We'll write the logic after we define our struct

for this context.

So we can go back down here and define that.

Now that we have the instruction,

we're going to define the struct for create entry.

So pub struct CreateEntry.

When you are defining your context data structure,

you need to define all of the accounts

that are going to be passed through the given instruction

that you're writing.

So the first account that we're gonna need

is the journal entry account, and that is for the state

that we just defined earlier.

So to do that you can use the Anchor account macro

and then we're gonna add some constraints on the account

for how we want to handle this account

throughout the instruction.

So the first thing we wanna do

is we wanna initialize the account.

We're initializing the account because it doesn't exist yet.

So we'll use the init constraint.

And then we want to have a PDA on this account.

So we're going to define the seeds for this PDA.

So there's about infinite ways

that you could define a PDA.

Here, I think that the best way for us to do it

is specifying the owner and then another parameter.

And that's because we wanna know

who each journal entry belongs to.

So having the owner connected to the PDA is a really easy way to derive that.

If we just had the owner, the owner wouldn't be able

to create more than one entry on Chain.

We need to add another parameter to allow them

to create more than one entry.

So we can do this by adding the title.

So if we add the title, the owner can go in

and create multiple entries based on the title.

We're just gonna add that for our seats.

We'll do title as bytes,

and then we'll add in the owner's key as ref.

And now our seeds are defined.

Anytime you define a seed, you always have to add the bump.

So we'll add that here.

And now since we're initializing an account,

we have to be able to calculate the space

that it's gonna take up on Chain.

This is needed so we know how much rent that we have to pay.

So we're going to define the space.

And whenever you're calculating space,

you always start with eight.

Eight is the Anchor discriminator, so we'll add eight.

And then for the rest of the values

in the given journal entry struct,

we can just calculate that by using a InitSpace

since we use the derive InitSpace macro here.

So we'll do a journal entry state InitSpace.

Okay. And now lastly, we just need to define

who's gonna pay the rent.

So we need a payer and we'll make the payer the owner

of the journal entry.

And then all that's left here is to name this account.

So we'll make this a journal entry of account,

and it's going to be using the general entry state.

Now that we define that the payer as owner,

we haven't specified who the owner is yet.

So we're gonna do that next

as the next account in our create entry direct.

So we're going to once again use

the account macro from Anchor

and we're going to specify who the owner is.

So in our case, we want the owner to be the person

who's signing the instruction,

so it's just gonna be the signer.

Now if you look at how our accounts are set up,

we have the owner being the payer

when we're initializing the journal entry.

Because the owner's going to be paying,

it's gonna be changing the state of the owner's account.

When that happens, we have to define

that the account's mutable.

So we're gonna add the mute constraint here on the account.

And this is just because in Rust all variables

are immutable by default.

So whenever we want something to be mutable,

we have to specify it.

And then lastly, we just have to define the system program.

So system program and we're good to go.

So now you can see we have one error showing up

on the account, and that's for title.

Now, why is that happening?

It's because we haven't specified what the title is yet

and we don't know what the title's gonna be.

So the title has to be a user input parameter.

So we're gonna go back up to the instruction

and we're gonna add a parameter for title,

and that'll be of type string.

Okay. And now this is being passed through the instruction,

but it's not being passed to this create entry struct.

So to be able to get this data down here,

we're going to have to specify that you're pulling the title

from the instruction.

And we can do that by using the instruction macro.

So instruction, and we're passing through the title

that's coming through from the instruction

user input parameter.

So now we're good to go for everything needed

for our context data stricture

for the create journal entry instruction.

Now we can go in and start writing our logic.

So if we look at the instruction,

we're passing through the context,

always the first parameter, we're passing through a title,

which is a user input value.

And then there's one more thing

that the user's gonna be inputting on the front end, and that's gonna be the message for their journal.

So we're gonna add that as another parameter,

and that's everything we need there.

So all that's left is just defining the logic

for the instruction.

And what are we doing when we're creating a journal entry?

We're just saving information back to the account state.

So that's all we have to do here.

To do that, we're going to load

in the journal entry account.

So we'll let the journal entry equal,

and we're making this account mutable

because we're going to be changing information

in the account.

So mutable, and we can load that in through the context.

So we'll do context.accounts,

and the account that we want here is the journal entry.

So you can see here, this is the journal entry account

that we're loading in through our context.

And Copilot knew what we were doing

and just finished everything.

So we'll talk through what Copilot just did here.

We have our journal entry, and then all you're doing

is updating the fields on the journal entry.

So the journal entry owner,

we're referencing the owner's key.

And the owner, as we've defined here is the signer

of the instruction.

We have the title and the message,

which are both user input parameters of the instruction.

And that's everything we need to save.

So our very first create instruction is completed,

so we can move on to the next.

So a CRUD app, create, read, update, delete.

Read is just querying the blockchain.

So we're gonna move on to update,

and we're going to do an update journal entry.

(keyboard clattering)

Okay. So as you can see here, Copilot is really great

and already knows what needs to be done

for this instruction.

So I just tabbed over through the Copilot suggestion,

but we have the update journal entry.

The very first parameter that's being passed through

is the context.

Now we need to define what this data structure

is for the context.

So let's go down here and do the same thing we did earlier,

but now we're going to be doing it for update entry.

So derive accounts macro,

and then we're going to do, oops, update entry.

(keyboard clattering)

Okay, here with the data structure,

you want to define all of the accounts

that are correlated to the instruction

that you want to have the data structure for.

So we need to pass through the first account.

That account is going to be the journal entry account.

So let's define that here and we'll use the account macro,

and we'll add some constraints to it.

So here the constraints are gonna be a little different.

It's already been initialized,

so we don't need the Init constraint,

but we're going to be changing the state.

So we need it to be mutable.

So we'll define mute as a constraint,

and then you always have to add in the seeds,

so it's able to derive the correct PDA of the account.

So we'll do the seeds and the bump.

And here the seeds should be the same as above title

and the owner's public key.

And now there's one more thing that we have to do.

As we talked about earlier, the rent for storage on Chain

is calculated based on how much space it takes up.

So if we're changing the length of the title

or the length of the message,

it's going to change the amount of rent

that a user needs to pay for space.

So let's say we had a message

that was only 10 characters long and I want to go in

and make it a hundred characters long,

I now have to pay more rent and vice versa.

Let's say I deleted a hundred characters,

now I'm paying less rent

and I wanna get those extra lamports returned back

to my account.

So to do that, we're gonna use the realloc constraint,

and this is just reallocating space

for a specific account on Chain.

So we're gonna calculate this space here.

So we'll do realloc equals

and have the new space calculated.

So eight is the discriminator,

and then we'll do the journal entry INITSPACE.

And then whenever we're doing this, we still need to specify

who is going to either receive the extra lamports

that are being returned or pay more lamports

that need to be given for more space.

So we'll do realloc payer.

(keyboard clatttering)

And that payer is once again gonna be the owner.

And there's one more thing.

So we can do realloc zero,

and what this is doing is just setting

the original calculation of space back to zero

and then recalculating everything.

So we'll set realloc zero to true.

And then lastly, we're just going to define

what this account is.

So it's our journal entry account.

Okay. Now if you remember from before,

this data structure doesn't know what title is,

so we're going to have to do the same instruction title.

So we'll pull that macro in,

and then we're going to add that as a parameter here

for updating journal.

So title is also going to be a user input parameter

for the instruction.

So now we specified the owner,

but we need to pass that account through as well.

So we'll add that in, same thing as we did earlier.

So it needs to be mutable because it's going to be paying

and it'll be the owner and the owner is the signer.

And then the very last thing we have to pass through

is the system program.

And now our data structure

for our update entry context is completed.

So we can pass that through here

for our update journal entry instruction.

If you look at the instruction,

we have this little warning under title.

And that's happening because title

isn't actually being used in the logic of this function,

this is just a little Rust thing.

We're going to just add an underscore here to say,

hey, I know that this parameter

isn't being used in the logic,

but I need to use it somewhere else.

So I still need to take it in.

Everything here is done.

We have our update journal entry.

We're passing through both the title and the message.

We're updating the message in our journal entry state,

and we're passing through all of the corresponding accounts

in our update entry data structure for our context.

Creating the journal is done,

updating the journal is done.

So all we have left is to delete a journal entry.

So we can go in here,

and we're going to write out our instruction for deleting.

Okay. So this time Copilot has been doing pretty well,

but this time it's not correct.

So we're going to delete what it suggested here,

and just have this instruction returning an okay value.

And that's gonna happen because all of the logic

for deleting an account on Chain

is going to take place inside

the delete entry data structure for the context.

That's because all it's doing is handling accounts,

and all of the account handling takes place inside

the data structure.

We'll go down here and make another struct

for deleting an entry.

(keyboard clattering)

Okay. First account we're loading in,

just like before is our journal entry account.

So we'll do the account macro.

We're changing the account state, so it's mutable.

We need to specify the seeds and the bump,

so it's able to derive the correct PDA.

So seeds, the same as earlier, bump.

And then there's one last constraint we're gonna add and we're gonna add a close constraint.

So what close does is if you run this instruction,

it's going to close the account,

but it's only gonna close the account if the public key

that I specify the close to be equal to

is the signer of the instruction.

So I want the person that can close this

to only be the owner of the account.

So close equals owner.

And we'll do, define the JournalEntry,

and then we're gonna specify who the owner is,

which is the signer.

Okay. And then lastly, we pass through our system program.

And then one last thing to pass through is the title,

just like we did earlier.

So we'll pull this macro down here

and then make sure title is passed through which it is.

And look at that. Our program's done.

So if you go here, you can see the create update

and delete entries are completed

in our program instructions.

And you can see this little warning here for context.

And that's coming up just like earlier with title,

it's because context isn't being used in the logic.

So we're just gonna add an underscore here and say,

hey, I know that, but I'm using it somewhere else.

And then we have all of our context data structures

for create, update, and delete.

And then we have our state,

and that's everything we need for a CRUD app.

So we'll go in and we're going to build the program.

So if I cd into my anchor directory,

I can just run anchor build,

make sure everything builds correctly and compiles.

Okay, so now you can see our program compiled,

everything finished and we didn't have any issues.

If something goes wrong within your program,

the Anchor compiler will be able to pick it up,

give you a warning, show you how to fix it most of the time,

but we have everything correctly done and built.

Now that, that's built, what happens is it builds

and sends an IDL to your target folder here.

So you can see that this is my new IDL,

and this is named a counter still

because I did not update the name in the Cargo.toml.

So we're just gonna go in there and update it to crud-app.

So we're just rebuilding with the name CRUD

just so we can have everything named the same

throughout the application.

Okay. Now we're good to go.

So once we build the program,

we're able to now deploy it on Chain.

Throughout this bootcamp,

we're going to be using everything local.

So I'm going to open up my terminal

and run a local validator

so we can deploy to our local host.

So here's my terminal.

And to get your local validator running,

you're just gonna run Solana test validator.

(keyboard clattering)

And we're now initializing an instance of the local host.

And you can see that the validator is running

on my local host right there.

So now that this is running, I can go in to my terminal

within the crud-app directory and do anchor deploy.

And I'm going to specify that I want it on local.

So we're gonna do provider.cluster localhost.

Okay. It's actually localnet for anchor.

So if you're using the Solana CLI, it's local host.

If you're using Anchor, it's localnet.

That's one fun fact to remember there.

So you can see it's deployed.

We have our program ID here.

One thing to always double check

is to make sure all of your program ID keys

are synced throughout the workspace.

So now that it's deployed,

we're just gonna run anchor keys sync,

and you can see all program ID declarations are synced.

Now that it's deployed, we're able to connect this program

to a front end.

So let's go into...

We can close out the Anchor folders

and we're gonna go into the web folders.

So in our web folder in the NPX Create Solana dApp Scaffold, we have app, we have components,

and we're gonna be working between those two folders.

So first we're gonna go into components,

and we're going to go to counter, we'll go to data access.

So we're just gonna make some updates here.

This was previously created for the Anchor counter program

that comes with the scaffold.

So this is just updating,

it's fetching all of the information from the state.

So we're changing the state from counter to journal entry,

and we're gonna do that everywhere.

You can see that we have instructions from the counter,

like close and increment and decrement.

All of those can be deleted, and we're gonna rewrite them

for the instructions for the CRUD app.

So we'll go in here, the close mutation can go away,

the decrement mutation can go away, and you can see

that it's returning an error

because it's not finding these instructions

in the corresponding IDL.

So TypeScript knows that something has changed and we just have to update those changes.

Okay. There you go.

Now we can just go in here,

and we're gonna do our create entry

and we're going to userMutation.

And now we're going to be collecting information

from the front end.

That's a user input value.

So we're gonna need to specify what's being collected.

We're going to need to make this an sync await,

so it's able to take everything from the front end

and then pass it through.

So we'll do.

(keyboard clattering)

We will set up our create entry args,

and that's gonna have to be an interface

that we're gonna define outside of this.

We'll do an interface for CreateEntryArgs,

and that is gonna be the title that we need to pass through,

which is a string type.

The message is also a string.

Then the owner, which is a public key.

This initialize can also go away.

Everything that's related to the counter can just be erased.

Okay. So since creating the entry

is part of the overall journal program

and your initializing a new account,

we're actually gonna have that in the use program.

So we can go in here and do const createEntry.

We're going to use the mutation with the CreateEntryArgs.

We need to specify the mutationKey.

Oops.

This is just a journal entry.

And then we have create, and we have the cluster.

And then we'll define the function.

And this is gonna be an async function.

And what we're grabbing from the front end

is going to be the title, the message, and the owner.

And now all we're gonna do is return the program method

for creating a journal.

We can delete this return that happened earlier,

create journal entry,

IPC to send the transaction.

Okay? So that's what we're doing.

We have the program methods,

we're calling the create journal entry instruction,

and we're passing through the title

and the message from the user input parameters

that we're grabbing from the front end.

Now, since this is an async function,

we just want to have an on success and an on error.

(indistinct) success.

(keyboard clattering)

So if this is successful,

it's just gonna sign the transaction.

And then refetch all of the accounts for the program.

And then if there's an error,

we'll add the logic into that.

And just make sure we return the error message.

Okay. So that's everything we need for create entry.

And then we just need to return that.

So we are returning our program,

our GET program accounts, and our create entry.

So everything's set up to be able to use the program.

Now we want to go into using a specific account.

So what this used program is doing

is just initializing a new account.

If you want to update an account that already exists,

we need to derive it from the PDA and then be able to go in

and update it.

So that's gonna be the second function here,

which is using an account.

So we'll update that.

We're able to query specific accounts,

and now we want to be able to update an entry.

Okay, so once again, we're taking the user input

from the front end, and adding in the CreateEntryArgs

that we specified earlier.

And we'll need, oh, let's see.

Let's see what Copilot did.

So we have the mutation key for the journal entry.

We're specifying that it's update,

and we're using the cluster.

Now we need to specify the function.

It's an async function.

We're taking in the title and the message, and the owner from the front end.

And then here we're doing update entry.

And actually in this case, we don't need the owner,

so we can delete that.

And we're just taking in the title and the message.

And then we want to have an on success and an error case.

And that's defined here.

If it's successful, it's going to sign the transaction.

If it's not, it's going to return the error

and looks like that's good to go.

Okay. And then last, we just want to do the delete entry.

So we'll add another const in here for delete entry.

Okay, let's see what Copilot did.

We have the mutation key.

We have journal, delete and the cluster. It's an async.

However, yeah, we don't need to take anything in

from the front end.

We're gonna take the program.

This should be program methods, not RPC.

Copilot was close.

Delete journal entry.

And we're going to need to take in the title from the front end.

And the title's needed to be able to derive the PDA,

so we'll take that in.

(keyboard clattering)

Okay. So we have our function of the delete entry.

It's taking in the title as a parameter.

That's part of our async function here.

Actually, there's a way we can do this

without making it an async.

If the card already exists on the front end,

it will have what the title is.

So it can just pull that without needing

to be in user input value.

So we can remove the need for an async here

and just have the title.

So the title will be already on the front end.

You'll have it on the card, and when you click delete,

it'll know which journal entry it's corresponding to.

So we have the title, the program methods, delete entry,

passing through the title here.

And then there we go.

So we have our signature,

on success it's going to be a transaction

and accounts refresh.

We don't need an error and we actually don't need this.

This is what Copilot generated,

but since we're only passing the title,

we don't need the CreateEntryArgs.

So yeah, we're good to go there.

Now all we need to do is return these constants.

(keyboard clattering)

And now everything's updated on the front end.

So in the front end we have things separated

by accessing data through our smart contracts.

All of the data access is completed.

Now we just wanna update that data on the UI side.

We'll go into the UI file,

and we're going to update all of the information here.

So let's see. We still need our key pair connected.

We won't need the memo. Get rid of that.

Okay. So we're gonna change a few things here.

So this first counter card,

it's actually going to be for creating a journal.

So we're going to want to have the user input fields here.

This is where we're actually going to grab the data

from the front end.

So we're going to want to have our title

and a way to set the title.

And we'll import useState from react.

And we'll do the same thing for message.

And then we're gonna wanna use our create entry instruction

that we just created from the data access file.

So we'll pull that in.

And then we also want to be able to use a public key, so.

And this we're gonna use from useWallet.

And this is coming from the wallet adapter library.

We have our front end, we need to have the title

and the message be a user input.

So we wanna make sure that these values

are actually input correctly from the user.

So we're just gonna have a check here.

So what we're gonna do isFormValid,

and that's gonna make sure that the title exists

and the message exists.

And then we want to be able to handle the submit

of the information on the front end.

So we'll make a function for that.

So if there's a public key connected on the front end

and the form is valid, then we're gonna be able to execute

the create entry logic check.

Okay. And that's all we need there to handle the submit.

So we need to do one more error handling,

and that's gonna be, if there isn't a public key connected,

we need to return that the user needs

to connect their wallet.

So we'll do an if there's no public key,

we will return:

Connect Your Wallet.

Okay, great.

So now all we're doing is just handling

the UI side of things.

So when we return, we can delete everything

that came with the counter

'cause we are not gonna need that.

Okay, so when we return, we want to have an input for text,

we want to have a text area for the message, and then we wanna have a button for someone to submit.

So we'll do a div, have an input,

and that input is going to be of a text type.

Oh, there we go.

So Copilot has it.

We need the text type, we need the placeholder

is gonna be the title, the value is gonna be title

and on change, it's going to set the title

by using the use date from react.

So that's good for the input.

And I'm just gonna add in the class name actually,

just so it sticks with the styling

for the rest of this.

We have our input, input bordered. I want it full.

And I also want max width extra small.

There we go.

Okay, so input is done.

Now we need a text area input as well.

So we'll do text area,

and then Copilot has it.

We have placeholder message.

The value that the user's gonna input is the message.

On Chain, it's gonna set the message in the state.

And then we have our class name.

So we have a text area, text area bordered,

and then I also still wanna add in the max.

There we go.

So text area is done.

And then lastly, we need a button.

So we have our button and let's see.

On click, we want to handle submit.

We want to have it disabled if the form is not valid.

And we also wanna add in if createEntry is pending.

Oops. So is pending or isFormValid,

it'll be disabled.

We have the class name, is going to be a button.

I don't want it to be a primary button here,

so let's change that.

So we'll do a small button.

Okay. There we go.

Okay. And then we just need to close our div,

and then this section is good to go, so.

(keyboard clattering)

okay, so this is done.

So what this is doing here is the card to be able

to create new journal entries.

Now we just need our last UI update,

which is going to be the card of a specific journal entry

and how to update a specific journal entry.

So that's our create.

Now, we'll go to the list.

So in the card you can see it has

all of the counter information from before,

so we're gonna delete that and we'll delete the memo

'cause we don't need it.

Actually, we'll delete all of this.

Just get rid of it. Okay.

Great. So in the card we're passing through

the account query.

We're gonna pass through update entry and delete entry.

And that's going to be from the program account.

And it's going to be passing through the account.

And now we're gonna do...

We're going to have the public key.

So we'll do the useWallet just like we did earlier.

This is public key. There we go.

Okay. And now just like we did earlier,

we need to set the message

because a user is gonna be able to go in

and update a message.

So they need to have that space to update it,

and we need to be collecting it on the state.

So const message and setMessage.

And we'll do useState, okay.

This was the optimization we talked about in the contracts.

It doesn't need to be a user input value,

we're actually just querying it from the account.

And the title cannot be updated,

it's just there because it's part of the PDA for an account.

So we'll just add in title

and that's gonna be an account query.

Okay. And I spelled const strong,

so we'll fix that. Great.

So we have our public key, our message, our title,

and then we'll do the same is formValidCheck.

So we'll just copy that over,

we don't need to retype it.

But actually here we're not having a title,

so we'll just update this to only be for message.

Okay. And then we also need the handle submit,

so we can copy that as well.

Just some less typing and we'll need the public key.

So while we're up here we'll copy that as well.

So here we go.

Handle submit publicKey.

And now, so delete this,

everything related to the counter, we're just deleting.

Now we can do our return.

And this is going to be an account query

and it's just gonna load all of the accounts associated

with the program.

This is just gonna be a lot of UI work.

So we're going to span, className loading, there we go.

And then else we're going to return the card.

So let's actually see what Copilot did.

So we have the class name card.

We're gonna just make this a little fancier.

So let's make it a bordered card, border base.

Let's do like 300 and then.

(keyboard clattering)

Spelling things wrong.

Okay, so class is done,

and now we're going to do the card body.

We'll just center things here.

Okay, we'll do this a little differently.

Okay, so we're going to do the className.

We'll just add a space in here.

Okay. And now we have our header. We have a card title.

We'll center the card title. Okay.

(keyboard clattering)

And now we're also just gonna wanna add in a refresh here.

So onClick it's going to refetch all of the account queries.

So onClick, we are gonna do accounts query refetch.

There you go.

And if you're following along

and see things auto completing here, it's just Copilot.

Gonna change what Copilot did a little here.

So this is actually gonna be an account query for the title.

(keyboard clattering)

Okay. Clean up some code a little bit.

So we have our title here for the card.

And now we have the title,

we're gonna wanna add in the message.

So we'll just add that underneath.

So let's delete this Copilot recommendation

and just add in the message.

And this will be an account query as well.

Okay, great. So that's done.

Now just visualizing the card,

we have the title at the top of the card,

we have the message displaying.

Now since we want to be able to update a journal entry,

we wanna have some space for a user to type

in their updated message and a button to click that.

So we're gonna add that in.

It looks like Copilot did a pretty good job here.

So let's just go over.

We have our card actions, we'll just add a justify here.

Just clean up the UI a little bit.

Okay. And then we have our text.

We change this to a text area.

Text area is gonna have a placeholder for message.

It's gonna have the message value,

onChange it'll set the message.

And then we'll do a class name.

And there we go.

So there's our text area and then we have the button

to handleSubmit for the updating and entry.

And that's gonna be labeled update.

We can just specify a little more update journal entry.

Okay. And then we have delete entry.

So this will be a delete button,

and we can just do an account query here for the title.

There we go.

And now since title could be undefined,

we're just gonna add in a little check here

for error handling.

So we'll do this and we're gonna do if.

(keyboard clattering)

So we will just change this a tiny bit.

Okay. So we'll do a const.

There you go.

So if it exists,

then it'll actually execute the instruction.

Okay, clean that up a little bit.

Okay, so now we'll clean up our imports that we don't need.

Okay. And then we need to find the name for createEntry

'cause it looks like we didn't name it that,

so let's go back

and see we have update, delete, we have createEntry.

So let's just make sure it was imported.

Okay, so this should actually be updateEntry,

there we go.

And then we're gonna have to do the same thing from earlier

because title could be undefined.

So we're just going to add in a check there.

So if title exists then we will actually do this,

so and title. There we go.

And it looks like this is good to go.

So one more check.

We're gonna check in on the feature.

We just need to make sure we are returning program ID,

which we're not.

So we'll add program ID in here, and we're good to go.

So let's try to build this.

So we're gonna navigate into our web app

and we'll run npm run dev,

and see what we've got going on here.

So let's open up a browser, and we'll go to our local host

and you can see our web app

is now running on our local host for 3000.

And here's our app.

So this is what the front end looks like

when you're using NPX Create Solana dApp,

you have this basic UI, you have the accounts,

this shows you your connected wallet account.

So this is my Devnet wallet.

You have your clusters.

Right now I'm connected to Devnet,

but I built and deployed my program on localnet.

So I'm gonna change that connection over to local. Okay.

And we're just gonna request an airdrop,

make sure I have funds on local.

Then we can click over to our program.

And you can see here, program not found.

So we're gonna go back here

and just make sure we have everything correctly updated.

So we'll open a new terminal,

we'll cd in the anchor, we'll do the anchor build

and then anchor keys sync, make sure all of the program ids

are connected across the workspace, so.

And then we'll just do anchor deploy,

specify the provider cluster to be a local host.

So it deploys the local.

localnet for anchor.

Local host is Solana CLI.

Okay, so here's our program id.

Let's just refresh this.

Okay, so you can see here the program ids don't match up.

So that means something in our data access

is pulling an incorrect program id.

So it's probably coming from the counter.

So we're gonna go back over here to our use counter program.

And where is it finding the program id.

So gets ID, cluster, devnet, public key.

So this public key needs to match the public key in our Anchor program.

So we'll go back to our Anchor program

and just make sure it's connected.

So see how they're different,

that's where we're going to update.

So in the NPX Create Solana dApp,

the one part that doesn't update with Anchor keys sync

is right here.

So you go to your counter exports

and you just wanna update, oops,

you wanna update the public key to the new public key.

Okay, so now let's see if this updated.

We're going to close out this instance

and just rerun our little cd back to our web

and then we'll run npm run dev.

Okay. So now we're on our local host. We'll go here.

I'm gonna refresh. Okay.

So we're just gonna find where it's generating this account.

So I'm gonna copy this over, and command Shift F

and search this.

Okay, so here it is.

This is what we also need to update.

So here you have these type scripts,

and it has an address that is our old public key.

So we'll go in here, take in the public key that we want,

that we actually deployed.

And I'm going to go into, let's see, how do we do this?

We will just paste this here so I have it.

We'll go back here, copy this, ran shift F.

We're going to paste,

one we don't want we'll click in here

and then we will paste this one in. Okay.

Make sure all of this goes away. Save. Okay.

Now we'll go back over here, refresh.

And here we go. We have our updated public key.

So this is our account,

now we're just going to create a message.

So I'm gonna do Bootcamp,

and I'll make the entry learning anchor.

And we'll just submit that.

It's going to pop up your wallet to sign the transaction.

So I'll confirm. And there you go.

So there's my entry.

It's titled Bootcamp, my message is learning anchor.

Now I have the option

to either update a journal entry or delete.

You can see the update is disabled

because I don't have the correct information in the message.

So once I start putting something in that field,

it'll pop up.

So let's update it to, I learned anchor

'cause now I know what I'm doing.

And now you can see the button shows up.

So I have the ability to update, update, my wallet comes up,

sign the transaction, we'll refresh.

And now we have a new message.

I learned anchor.

And now last thing, delete.

So if I don't want this anymore, I'm gonna delete it.

Now it's gone.

So that's everything, we created,

we updated, we deleted, and we're able to read on the UI.

So we have our fully working CRUD app,

we have it connected to the front end,

and we're able to play around with the UI and test everything.

So we did it. Good job guys.

- Let's talk about tokens and also make our own token.

So far the only token we've discussed

is SOL which we use to pay transaction fees

when making our favorites and voting contracts.

SOL is the native token of Solana,

which means it's built into Solana itself.

But if you use blockchain in the real world,

you've probably used many tokens other than SOL.

On Solana, these are called SPL tokens.

If you're wondering about the name,

SPL is the Solana program Library.

A bunch of smart contracts which include the program you use

to create and transfer tokens.

For the rest of the course,

we'll just refer to SPL tokens as tokens.

Tokens are used to represent everything apart from SOL,

there are tokens for stable coins like USDC,

NFTs and real world assets like equities, metals,

commodities, dog tokens, and nearly anything else.

Tokens are produced by token mints.

Token mints are factories that make a particular token.

The factory is run by a person called the Mint authority.

The Mint authority must sign all the transactions

that mint new tokens.

Each distinct token on Solana has its own mint.

For example, when I purchased my beer earlier,

I paid with USDC, a popular stable coin.

A token whose value is tied

to a FI currency like USD.

USDC is made by a company called Circle

and they have the token mint address published

on their website.

The company Circle is the mint authority for USDC.

When we create our own mint with the token program,

we'll set ourselves as the mint authority.

The mint authority can mint,

I.e create new SPL tokens into any address

including their own.

When an account receives the tokens,

it'll need somewhere to store these tokens

since regular accounts only store SOL.

So it'll create an associated token account to store that.

An associated token account is just a PDA

with the seeds made from the wallet address,

and the token mint address.

For example, I can always find Alice's USDC token account

by looking up the PDA based on her wallet address

and the USDC mint.

And I can always find Bob's USDC token account

by looking up the PDA based on his wallet address

and the USDC mint.

And I can always find Alice's dog token account

by looking up the PDA based on her wallet address

and the dog token mint.

You get the idea. Let's make a token.

So let's make a token.

We'll be using the command line tools here,

but the concepts are exactly the same as if you're using

the JavaScript or TypeScript tools.

We'll also be using the new token program called

Token extensions that features a bunch of useful features

that we can add to our tokens.

One of those is the metadata feature,

which allows us to have our metadata.

All the things you see about the token, like a name

and description directly inside the token mint.

And if you dunno what that means, that's fine.

All it means is that we'll be taking less time

because we have less things to work on.

So let's open a command line.

Let's see that we have Solana command line installed, great.

Let's make a new directory called New token,

and let's change into that directory.

We are now in a big old empty directory

and the first thing we're gonna need to make

is what's called a mint authority.

If you think of a token mint,

like a physical mint that prints tokens,

you can think of the mint authority as being the boss

of that mint.

They're the person that needs to sign all the transactions

to mint new tokens.

If you are not the mint authority,

you can't mint tokens for a given mint.

So let's make that, we're gonna make a key pair

for our mint authority and we're gonna run

the command solana-keygen grint,

and we're gonna say --starts with

and we'll say bos, BOS and then one

and I will, I'm not a good typist,

especially when I'm recording.

Solana-keygen grind starts with Bos one

and if you give it a moment it will go

and make a keypair file, so.

There we go.

Our file is called Bos something.json,

whatever the public key is that matches

the private key inside.

So we now have our boss something or other.json file.

Let's make Solana use that keypair

to sign all our transactions.

So we do solana config set --keypair bos something.json,

whatever your JSON file is called

and it'll respond immediately.

And we can see that our keypair path has now changed to be whatever our BOS file is.

At the same time, we should also make sure

that the Solana command line tools are using Devnet

'cause that's where we're gonna make our token.

That is solana config set --url devnet.

And you can see in the response

that the settings now updated to use Solana devnet.

We can also double check that

if we just do a solana config get,

at any time we can always check

what keypair file and what network we're connected to.

So we've created our mint authority,

but because this account's gonna be writing

to the blockchain, they need some SOL

to make those accounts.

So we have our bos something or other is the public key

of our mint authority.

Let's go to say faucet.solana.com,

paste in that wallet address and look,

you know, one SOL is fine, 0.5 SOL is fine.

And we'll ask for that SOL on Devnet

and we can get some free Devnet SOL.

Excellent, that says airdrop was successful.

And if we do solana balance,

we can see that our balance is 0.5 SOL.

So we've made our mint authority,

which is gonna be the bos of our token,

the account that has the permission to create new tokens

or mint new tokens.

And we've given our mint authority a little bit of SOL

to make new accounts.

Great. The next thing to do

is to actually make the token mint.

So we're gonna give our token mint a fancy address.

It's just gonna start with mnt.

Just so we can remember it's our mint.

So we will run solana-keygen,

and we will say grind, an address that starts with mnt.

We need one of those addresses.

Give it a moment, and it will create an address

that starts with mnt something.json.

So that address is where we're gonna put our token mint.

Let's put a token mint at that address.

That is spl-token create token.

And we're gonna specify the program id

of the token extensions program,

which starts with token and then a Z

and then all this other stuff.

So we'll just paste that in.

And we want to use the metadata extension

'cause it makes our lives easier, enable-metadata.

And we will finally, we will say our address

to use to make our new token.

Great. That's done.

So we've now made a new token on Solana.

The address of our token is the same

as the one we we ground earlier.

It has nine decimals,

which means that if you've ever transferred SOL before,

you'll notice that to send one SOL,

you send 1 billion lamports.

It's the same with our new token.

We've specified nine decimal places.

So you send 1 billion whole mining units

to send one major unit.

This is actually not even a blockchain thing,

it's a financial programming thing.

Whenever you send someone $2.50 in the real world,

you probably actually send them 250 cents.

And it's because computers don't handle decimal very well.

They work in binary.

So if we always send whole numbers

of whatever the mining unit is, cents, lamports,

your own tokens mining unit, it makes it easier

for the computers.

So we've made our new mint,

let's go check it out in Solana Explorer.

So I'm in Solana Explorer

and I will say Devnet, thank you very much

and I will paste in our token mint address.

And check it out, it says unknown token,

it says a token 22 mint, which is using token extensions.

There haven't been any tokens minted yet.

Nine decimals as we just talked about.

And there's not really that much here.

There's no kind of metadata or anything yet.

And it says unknown tokens.

So let's fix that.

Let's add some metadata.

Now to do that, you're going to need to upload firstly

an image and then a JSON file somewhere public

on the internet.

If we were making a token for mainnet,

a real serious production token,

we would probably wanna use a decentralized storage service

somewhere like Arweave or IRIS.

That is a basically kind of like a blockchain,

but for JSON files, and images and all that kind of stuff.

Rather than, you know, Solana is a blockchain

for transactions like traditional blockchains.

These decentralized storage services are designed

to keep things online for a really long time

'cause they're not owned by anybody and they're resilient

to attacks and all this kinda stuff.

They're a good place to put something

that you wanna be around for a really long time.

In our case, this is just a test token.

So we're gonna use GitHub,

and we'll just be using those raw GitHub links.

The important thing to realize here

is that our link has to just open our image file,

or our JSON file directly.

No HTML, no anything else around.

Make sure that when you visit that link in your web browser,

it just shows an image file or some JSON

depending on the image or the JSON.

So upload the image first, add it to a metadata file

that looks a little bit like this.

You'll just have a name, a symbol, a description

and the image that you uploaded earlier.

Take that metadata file and upload it

to somewhere public as well.

Make sure that you have a URL

that loads your metadata file directly in your browser.

For example, I have one right here.

There we go.

So I can now add that metadata to my token.

So spl-token.

And we're gonna say initialize metadata, and we're gonna specify our token.

I'm just gonna get it from the file name,

now mnt file name.

And we're gonna specify name for our token

and a symbol like.

And then finally that URL to get to that JSON file.

And then Solana will go and take that JSON file

and put that image, put that data,

I mean to say directly inside the blockchain.

You'll see everything has gone successfully.

And if we go back to Solana Explorer at this point

and reload our token,

you'll see unknown token has been replaced by Example

as we specified before.

And now there's a whole bunch of metadata, our name, symbol,

the URI with the image

that's all now showing on Solana Explorer.

So great, we now have a token mint and it looks nice.

So we can use this token mint to mint tokens.

Let's mint some tokens and as the mint authority,

we can mint them into anyone's account.

But let's mint some into our own account, so that we can transfer them out to other people if we like.

Spl-token, we're gonna need a token account

in our personal mint authority account

just to store these tokens.

So create accounts for this mint.

So we're just making a PDA or an associated token account

to store just these tokens in our own personal account.

Once we've done that, we can go and mint some new tokens.

That's spl-token mint,

our token, which is mnt something or other.

And how many tokens do we want?

I'd say 100, maybe 1000 tokens. So that's good.

Done. Let's go and look at firstly our token mint

and then our own personal account to see what's changed.

So if we go back to Solana Explorer

and look at the token mint again,

you can see the current supply,

which is the amount of tokens

that have ever been minted is 1000.

In fact, let's mint another 1000.

You can see Solana Explorer updates again.

And you can see 2000 tokens have been minted.

Also, let's check our personal account

as the mint authority, which we called, what was it?

BOS something or other. There we go.

And if we check in our own account,

we can see that 0.5 SOL that we gave to ourselves earlier,

minus a few transaction fees.

But more importantly, if we check in tokens,

we can see that we have 2000 of these tokens.

And the type of tokens is our token mint address

that we made before.

We can transfer these tokens, we can burn them,

we can delegate them, we can do all the normal things we do

with Solana tokens.

So there you go, you now have your own token.

I hope you enjoyed this

and I'll see you in the next chapter.

Now let's make an NFT.

And here's the good news.

You already know most of what you need to make an NFT.

NFTs are just SPL tokens,

exactly like we made in the previous lesson.

Just like other tokens,

they're minted from token mint accounts stored

in associated token accounts and transferred

with the token program.

However, NFTs have a few characteristics that are distinct.

Firstly, they're unique,

I.e each NFT only has one ever minted.

Also, they can't be broken up.

In other words, each NFT has zero decimal places.

Additionally, the metadata for an NFT contains links

to another file stored on a file storage service like Iris

or IPFS.

This off chain metadata file has links

to media files like images, videos,

or 3D objects, any traits the NFT has,

links to the project that made the NFT and more.

Authenticity is also handled differently with NFTs.

With regular tokens like USDC, we know they're authentic

because their mint address matches

the official USDC mint address.

Since each NFT has a unique mint address, we use the concept of collections to verify that each NFT is actually minted

by the right project and not a forgery on person created.

Let's go make an NFT.

All right, so let's make an NFT.

We're gonna start by making a folder called new NFT.

We will change into that

and we will run npm init -y

to create a blank npm project

at package.json and all those things.

We're going to install a couple of packages as well.

We wanna install from the metaplex-foundation,

a package called mpl-token-metadata

and also a package called,

also from the metaplex-foundation umi bundle,

bundle defaults.

UMI, by the way is the Japanese word for C.

You'll notice a lot of things in Solana like C level,

like anchor have names based on seas or oceans.

Cool, they're all installed.

I'll just clear my screen, and let's go make a new file

and we will call it create-collection.ts.

So we have our correct collection.ts here and we're gonna start oddly enough by making the collection.

As I mentioned before, NFTs unlike regular tokens

are unique.

So whereas normal tokens have multiple tokens,

many billions of tokens in some cases

or with a single mint address,

each NFT has its own mint address.

So there needs to be some way

to bind all those NFTs together and also confirm

that any new NFTs are actually produced

by the owners of that collection.

So that's why we have collections.

So we'll start by making that collection,

and we're gonna import a few things.

We'll take createNft,

collections themselves by the way are also NFTs.

FetchDigitalAsset. And what else?

MplTokenMetadata, MPL is Metaplex.

That's the company that maintains the metaplex NFT tools

from @metaplex-foundation/mpl-token-metadata.

That looks good.

It is just showing up as great

'cause we aren't using it yet.

We're also gonna import a few things from Solana Helpers

and I should install the helpers as well.

npm i @solana-developers/helpers.

So let's import a few things from that helpers file

that's airdropIfRequired.

GetExplorerLink, getKeyPairFromFile

from Solana-developers/helpers. That looks good.

There's a few other things as well.

We also want to create umi.

Our umi instance is just a way we can talk

to Metaplex's tools, we can talk to other tools at umi

as well. But nearly all of the time,

in fact people just use it to talk to Metaplex tools.

Metaplex-foundation, it is from umi-bundle-defaults

is the name of the package.

There we go.

We want a few things from Web3.js as well,

that's actually already installed for us

'cause it's a dependency of some of the other packages.

We want connection and we want LAMPORTS PER URL from @solana/web3.js.

Oh and I haven't spelled connection properly.

And oh, LAMPORTS PER URL,

LAMPORTS PER SOL,

I don't know why I thought LAMPORTS PER URL.

That looks good. Excellent.

So we're gonna create a connection.

Connection, and we'll connect

to whatever the URL for Devnet is.

So clusterApiUrlfordevnet,

and probably also wanna import

clusterApiUrl as well. Oop.

And we'll also want to import clusterApiUrl

from Solana Web3.

So we have our connection, we'll want a user,

so getKeypairFromFile.

If we don't specify the file to use,

it's just gonna use the id.json that's in our home folder.

So that sounds good.

And let's give ourselves some SOL if we don't have any.

So go await airdropIfRequired connection to who's public key?

The user's publicKey.

Oh I just realized I forgot the await.

There we go.

Await getKeypairFromFile user.publickey

and we wanna get one SOL Solana.

So one LAMPORTS PER SOL

if our balance ever falls below 0.5.

LAMPORTS PER SOL. That looks good. Excellent.

Let's also just console.log loaded user,

and then we'll just say the user public key

And probably wanna say to base 58 so it prints out

in the usual string of letters and numbers

that we're used to seeing Solana public keys as.

So let's make our umi instance.

This is gonna be as I mentioned before,

how we talk to Metaplex's tools.

So constant umi equals createUmi

and we'll create our umi instance just using

the same endpoint as our connection.

So it's a umi connection to Devnet.

We'll tell umi the programs we wanna use, which in nearly all cases, as I mentioned before,

umi is pretty much a Metaplex thing.

So the programs you'll probably want use

are mplTokenMetadata. Sounds good.

So mplTokenMetadata is actually a function.

So we go umi.use and then we run mplTokenMetadata.

So we're gonna make a umi version of our user keypair.

So umi has its own format for key pairs.

So we'll just call it umi user,

it equals umi.eddsa.createsKeypairFromSecretKey.

Lemme just say user.secretKey.

So this is just a copy of user

but it's just in the format that umi uses for secret keys.

Again, Metaplex has their own ways of doing things.

So these might be a little bit different

than the normal Web3.js ways we've been using

in previous lessons.

So we've made our umi keypair

and we'll go umi.use keypairIdentity,

and then umiUser.

And I think we need to import key pair, key pair, no and we need to import keypair identity,

so we can get that from Metaplex Foundation umi

and that looks good. Excellent.

Great. So we've console.log,

say Set up Umi instance for user.

I always like adding console logs

just you know if we're making command line tools,

it's nice to have a little bit of feedback.

So we've loaded our user, we've connected to Devnet,

we've given the user some Devnet SOL if they needed to.

We've made a umi instance so we can talk to Metaplex's tools

and we've set that umi instance up to use our user

as the default signer for any transactions. That's good.

Let's go make the collection.

So we're gonna do const collectionMint

equals generateSignerUmi.

So it's gonna make it keypair using umi

and this is our transaction,

that is going to create our collection for us.

Await createNft as I mentioned before,

collections themselves are NFTs,

they're just NFTs that point to other NFTs.

You can even do fancy things like collections

that point to collections.

Go check out Metaplex's docs for that.

And we'll say umi, and then the options for our collection.

Mint will be the collection mint,

which is just that keypair we made earlier.

The name, let's call it, sorry, My Collection.

Symbol: MC.

I don't normally like using symbols for connections,

but Solana Explorer says no symbol

if you don't have a symbol for your collection.

So that's the only reason I like to do it.

A URI, I'll talk about this in a second.

URI, pardon me, add that comma.

What else?

SellerFeeBasisPoints, which I will say

is a percentAmount of zero.

Seller fee basis points are a way for artists

to make money on secondary sales.

So if an artist sells an artwork to somebody,

then a couple of years later,

particularly as an artwork has increased in value,

that person transfers it to another person,

then the original artist can make a royalty

on each of those transactions.

That's actually a really good idea.

But these days we actually have newer,

and better ways of handling this

'cause these basis points on NFTs used

to just be enforced by marketplaces being nice.

These days, we actually have things that are built

into the protocol like programmable NFTs

and like transfer hooks that can be used

to provide a mechanism for artists to get those royalties

and enforce them so people can actually transfer

that artwork without paying the original artists.

Very cool. Particularly for people who make money

as artists for their living.

And yeah, if you're interested in that,

you should check out token extensions.

There's a lot of really good things in there.

SellerFeeBasisPoints and isCollection, let's say true.

Great and it looks like we need to import percent amounts.

That's percent amount.

Where do we get that?

It's from, oh Metaplex-foundation/umi keypair.

So ah, just up to keep your identity.

Let's add percent amount. There we go.

And that all looks good.

So after we've made our transaction,

we can do await transaction.send and confirm

and we'll use our umi instance to connect out

to Metaplex's tools on the Solana Devnet.

And that looks good.

So we've loaded our user, given them some SOL,

connected to Devnet, created our umi instances

and created our NFT to represent the collection.

We're going to actually fetch the NFT that we've just made.

So const createdCollectionNft

equals await fetchDigitalAsset umi.

And then we say collectionMint.publicKey,

yeah fetch the NFT for the collection at that address.

And then we are done.

So console, sorry, console.log.

Sometimes it feels like VS code is heckling me when I type

and I will say, what will I say?

Created Collection, I will pick a nice emoji for a box

'cause the box has the collection in it

and I like logging emojis.

The address is, and then let's use getExplorerLink.

And what do we want?

We want a link for the address

of our created collectionNft.mint.publicKey.

And we want a link for Devnet please.

So that looks good.

Let's just run through our script to go through it again.

We have our connection to Devnet.

We're loading our user from ID.json,

we're giving them some Devnet SOL if we need to.

We're creating an instance of umi

which we use to talk to various tools.

And we're saying the specific tools we want to talk to

are the MPL token metadata program.

In other words, Metaplex.

We're creating a umi version of our user

and saying that umi should use that

to sign all our transactions.

Just making a keypair, then creating an NFT,

which is gonna make a mint using that address

and that NFT is a collection NFT.

So you'll see isCollection true,

that is our collection NFT.

We're gonna send that transaction

and then refetch it from that address.

We can run this but first we need to talk about this URI,

so that URI should be a JSON file

that you've uploaded somewhere

and that you can access directly via that URI.

For this case I've just used GitHub's raw URLs.

So that's when you upload a file to GitHub.

You can click on it and see the raw URL.

What you want for that URI or URL,

they basically mean the same thing

is that when you load that in your browser,

you just see the JSON file,

you shouldn't see any HTML or anything else around it.

So if you look at my screen right now,

it's just straight up JSON.

All it has is a name, a description and an image.

Also with that image file, let me show you the same thing.

When I load that image, the image just loads itself.

No surrounding HTML, no anything else.

You'll need both the JSON file

and the image file mentioned inside the JSON file to be able

to load up directly without anything else around it.

Again, in GitHub you can just use a raw URL

which is fine for our purposes.

There's quite a few

of these permanent online file storage services.

You can pick the one that works for you.

But the thing that makes them permanent

is they're like Solana decentralized.

So they can store your JSON file

or your image online for a really long time.

They're not reliant on any individual cloud provider.

So when people buy your NFTs in production,

they can know that the JSON

and the image will be around online forever.

So we've got our image and we've got our JSON file and when we load either of those URLs we just see the image

or the JSON file.

Let's take the address we've uploaded our JSON file to

and paste it into our URI field for our collection.

So that looks good.

We probably also wanna install esrun

'cause we're about to run it.

So npm I esrun.

Esrun if you haven't seen it before,

it's just a really easy way to run TypeScript

on the command line.

You don't have to set anything up beforehand, it just works.

So now that we've installed esrun,

we can just run our TS file,

that's npx esrun, create-collection.ts.

So we've loaded our user, we've set up an umi instance

for that user and you can tell it's thinking.

It's gonna go make our collection for us. Beautiful.

So it's made our collection

and because we did that getExplorerlink,

it's even given us a beautiful link to click on.

So we can see in Solana Explorer,

make that a little bit bigger for you

that our collection has loaded, the symbol is MC.

The master edition is a way of saying

that this is a collection,

and you can see a bunch of different information about this.

There's only one of the collection,

there'll only ever be one and it has a bunch of other fields

that are common to regular tokens.

Things like a mint authority,

like who's allowed to make these, a freeze authority,

who's allowed to stop them being moved around

and an update authority who's allowed to change this,

which all just public keys.

The update authority you'll see is the public key

that matches our ID.json file.

And if you scroll down you can see

when this NFT was created in the history.

You'll see two minutes ago that, that transaction ran.

So that was our transaction running through umi.

Great. So we have our collection, let's make an NFT

that's a member of our collection.

And then let's also verify as the collection owner

that, that NFT belongs to that collection.

We're gonna make another script here

and look what should we call it?

We'll call it just create-nft.ts and I'll save that.

I'm going to just copy all the imports,

and probably the user and the airdrop

and the umi and the umi user

because we can reuse all of those.

We might not use all the imports,

but we can just delete the ones that we don't use.

So in create NFT, we've pasted in everything

from our previous script all the way up

into we set up the umi instance.

The next thing we'll want to do is save the address

for our collection 'cause we're about to use it

to make the NFT that belongs to that collection.

So let's do const collectionAddress equals,

we'll say new PublicKey, oops, PublicKey

and we'll get the address from our collection.

I'm just gonna use our explorer.

We could also get it from the command line.

New PublicKey and that address,

it's complaining about PublicKey.

Oh that's just 'cause it's unused, that's okay.

And it'll need to import PublicKey

from Solana Web3.js.

We could also, if you wanted to do it this way as well

because umi has its own format for public keys.

We can also just go PublicKey,

which is actually a function that comes with umi

and there we go. That also works.

If we do the the normal Web3.js method

of doing new publicKey,

then later on we'll have to convert it

to the umi publicKey anyway.

So we may as well which is actually done

with the same function.

So we may as well just use the function in the first place

and just go collection address equals publicKey,

the function and then the address of our collection

that we got from Solana Explorer.

So we've got our PublicKey,

let's go make our NFT, console.log creating NFT...

Oh sorry, the S code gave me a great suggestion

which I also don't care about.

Creating NFT...

The NFT has its own mint, as again I mentioned each NFT

has a unique mint address

'cause they're unique, generateSigner umi.

That's gonna be just a new keypair that we make.

And then let's actually make the transaction

to create an NFT.

So const transaction equals await createNft,

we'll say our umi instance

and here's the options for our NFT.

The mint will be the mint address we just made.

The name of our NFT will be My NFT.

The URI will be another off chain data file.

I'll just put ... in there for now.

SellerFeeBasisPoints same as earlier,

we'll just say percentAmountZero

and we'll add some information about our collection.

We'll say the key of our collection,

the publicKey of our collection is collection address.

Actually I'll tell you about that in a second,

and we'll say verified: false

just because when we first make this NFT verified

we'll be set to false.

In a second we're actually going to sign or verify

that NFT being part of our collection.

And that will change verified to be true for us.

Then we'll do await transaction.sendAndConfirm

using our umi instance.

So we've made a transaction to create our NFT.

We've said that it's a member of the collection

that we made a moment ago and we've sent that transaction

off to umi to be processed and make everything for us

on the blockchain.

Let's go fetch the NFT we just made.

So const createdNFT equals await fetchDigitalAsset.

And using our umi instance we're gonna fetch the asset

that it sets, the NFTs unique mint address

mint.publicKey.

And then let's make a pretty console log.

So console log and I'm gonna add a little artwork emoji 'cause that's cool and we'll say created,

pardon me, Created NFT Address is

and then we'll do getExplorerLink

for address using createdNFT.mint.publicKey.

And we want a link for Devnet please.

That looks good.

So that's our whole file.

I'll just go and remove any of these imports

that we're not using.

We ended up not using publicKey from Web3.js.

We ended up just using the publickey

from Metalex Foundation umi.

It's a little bit weird umi having different formats

for public keys from Web3.js,

but hopefully that will be fixed in future.

I think everybody's moving to the new web crypto style

of public keys so you know that will get better in future.

So finally before we run this script,

I wanted to talk to you about that URI.

So just as we saw with the collection

that URI needs to be a file, JSON file stored online and that when you open it in your browser,

you just see the contents.

So I'll open up another tab here and show you.

For an NFT, it's just a name, a description.

You can add a symbol if you want to.

I don't feel like having a symbol for this one and an image.

The image inside also needs to just load directly

when you load it up in your browser.

So it's this cool ball with energy bits, that looks nice.

So same as before

if you're making NFTs that you are selling in production,

use one of these like permanent online

file storage services.

That means that your JSON file

and your image will be online forever.

So I will paste in my uploaded,

oops, not the image file, I want the JSON file.

There we go.

And I will run our script.

So just clear my screen.

NPX esrun, then create NFT.ts.

Let's make our NFT.

So it's loaded our user, set up our umi instance

and it's making our NFT for us.

Give it a moment.

Great. We've made our NFT.

So if we click on that URL, I use a max.

So let's command click to open up the URL.

We can now see our NFT.

It has the artwork that is the artwork

that we uploaded before, this actually...

NFTs can have a bunch of different files.

This just happens to be an image,

but we could also use an MP4 video.

We could use a 3D object like a NFBX file or an OBJ.

The name of our NFT is the name from the JSON file.

We didn't set up a symbol so it says no symbol was found.

And yeah it's a bunch of other normal token options.

Like there's a maximum of one of them,

and there's only one of them supplied because NFTs,

if you look at them as being regular tokens are unique.

SellerFreeBasisPoints is 0% as we said earlier.

And you know, you can see that it was created a couple of minutes ago, two minutes ago in fact.

We didn't set up any attributes on our NFT.

And if you look on metadata

you can see a bunch of things like where the JSON file is,

which is where it got that image from.

And you can see the creator.

The creator zero,

the first creator is our address from ID.json.

And you can see the collection, which is the collection

that CH file, also that CH publicKey

that matches our collection that we made a moment ago.

You'll see that verify news zero,

let's verify that NFT as being part of that collection,

so that it turns that verified to one.

Also art tools, particularly wallets or galleries

or art marketplaces really like it

if you have verified set to one.

That used to be that people would run around,

and like make an NFT that was a fake NFT

that looked like it was a member of a real collection.

So having verified NFTs is a way for the collection owner

to say yes, this NFT is a part of my collection.

So let's do that now.

We're gonna make a new file,

and we'll call it verify NFT.ts

and 'cause I don't like typing,

I'm gonna steal everything from that first file again.

So just loading up our user, giving them Solana,

setting up our umi instance and yeah,

all the way to set up umi instance for user.

So I'll just paste that in, and to verify an NFT,

it's actually pretty simple.

It's only yet another just a single transaction.

So we've made our create NFT script,

and we've pasted in everything all the way up

to setting up our umi instance.

We'll need to set up our collection address.

I will steal that from our create NFT scripts.

And we also need to set up our NFT address.

NFT and what was our NFT address?

I'll get it from Solana Explorer

eight four something or other.

That sounds good.

And it looks like we need to import that publicKey,

that function which is part of umi.

There we go. Cool.

So we have just a pretty default umi script

with a collection address and NFT address set up for us.

So we've pasted in our collection address

and our NFT address into verify NFT ts.

The only thing left to do is actually make that transaction

that verifies this NFT is a member of this collection

and that will flip that is verified to true on the NFT.

So let's make that transaction cons.

Transaction equals await,

and it's called verifyCollectionV1

and it takes an umi instance and a bunch of options.

The options are the metadata and this is the metadata

for the NFT that we wanna verify.

We'll actually use a little helper,

it's called fineMetaPda, it comes with metaplex,

and we'll just give it the options.

Find the NFT, sorry, find the metadataPda for this mint,

which is our NFT address, nftAddress. Beautiful.

The collection mint, which is in our collection address, nice and an authority which we'll actually just say

is the user running umi.

So if you do umi.identity, that is our authority.

We could also say authority is umi user.

Same thing, it just means one says,

it's this particular keypair,

the other it says it's whoever is running umi,

which is also the same user account,

the one from our ID.json.

So do transaction.SendAndConfirm umi. Beautiful.

And then we'll do console.log

and let's use some cool emojis.

I will use a VL tick for being verified,

and we'll say NFT verified.

And let's give it the NFTs address,

NFT as member of collection.

And we can say collection address

and let's also say see Explorer at,

and we'll say getExplorerLink for an address,

which is the NFT address on Devnet.

That looks good. Excellent.

I'll just save it and format it, and everything looks good.

I can remove some of those unused imports.

Don't have to, but I just wanna keep our file nice

and ensure there's no dead code.

Let's go and run our verify NFT.ts.

I'll clear my screen and do NPX esrun verify-nft.ts.

Give it a moment to run,

and it looks like it's finished already.

That was really quick.

Let's go check our NFT.

And what we're looking for here is in the metadata.

There we go.

Verified is now set to one,

Explorer actually updated while I was looking at there.

I was worried for a second.

Yeah, cool. Great.

So our NFT is now verified as a member of that collection.

So now we have a beautiful NFT.

If we click creators, our wallet address is shown as being

the creator, that NFT is verified

as being a member of a collection,

which means that wallets and galleries

and art auction sites will show our NFT nicely

because it's important that NFTs are verified

as being parts of collections.

We have a beautiful collection and yeah,

we can transfer this NFT to people,

we can set it up on auction sites.

We can make Daps that use this NFT

to like token gate access to something

that can be a real world experience or a digital experience.

If you wanna learn more about Metaplex's tools

that we were using in this lesson,

the tool we were using is called MPL token metadata.

So if you google that, you'll find all the documentation

about that tool.

And if you wanna learn more about Solana,

I will see you in the next lesson.

Our next smart contract allows users

to offer an amount of one token to exchange

for a desired amount of another token.

For example, Alice wants hundred dog tokens,

is offering to swap 10 USDC.

Without our program, users would have to swap

with each other manually.

But what if Bob promised to send Alice a hundred dog tokens

but took the 10 USDC and ran?

Or what if Alice was dishonest, received the 10 USDC

and decided not to send the 100 dog tokens.

In their traditional finance world,

when Alice swapped with Bob,

there would usually be someone in the middle holding

the funds temporarily and taking a percentage

of the transfer for themselves.

But in blockchain this is sold

with a swap program also called an Escrow.

Our program will transfer the funds provided by Alice

into a vault account and store details of her offer,

like the amount and type of token she wants.

In another PDA, when Bob decides to take the offer,

our program will transfer Bob's tokens to Alice

and transfer the funds in the vault to Bob.

Now remember, in traditional finance,

someone would sit in the middle taking a couple

of percentage points of what Alice and Bob are offering.

But with our smart contract, Alice got a 100%

of the tokens Bob sent.

Bob got a 100% of the tokens Alice sent

and they got what they wanted without having

to trust each other.

Fees were only a fraction of a cent.

That's the benefit of decentralized finance.

Let's get started.

So let's make a Swap program.

First thing I'm gonna do is run Anchor init swap

and I'm gonna use the multiple files template.

Because this project is a little bit larger

than say our favorites program,

multiple files will split our program up across different

files and directories just to organize

our project a little bit better.

So --template equals multiple.

Give it a moment to install. Great.

I change into that directory and let's open up our editor,

and have a look around our new project.

So here's our project.

The actual anchor program is in the programs Swap directory

and the tight script tests are in the test directory.

Program Swap is also a little bit more interesting

than the normal single file layout.

There's still a Lib.rs, but helpers is specific

to each instruction handler,

living the instructions directory.

There's a state directory

for storing basically the structs

for the accounts you're gonna make.

Constance direct, a constance file I should say.

And yeah, it's just...

This will make things a little bit neater for us.

So the first thing I'm gonna do is add some dependencies

'cause we're not just using Anchor here,

we're also using anchor-spl,

which helps us do things like transfer tokens around.

So let's do that.

Let's pop open Cargo.toml

and just where it has the anchor-lang,

we're gonna change that

to look kind of like a little object.

We'll say version equals 0.30.1.

And we'll also turn on the features equals

init if needed.

Init if needed allows us to create an account,

add an address if it doesn't already exist.

We'll also add our anchor-spl and what version?

The same as our version of Anchor. So 0.30.1.

And we may as well just say the current version

of Solana program as well.

It was 2.0.3. Great.

So we now have our program with Anchor and SPL installed

and the init if needed feature turned on.

I'll just fix that typo. Great.

Something else is, we should probably also look

at setting up our constants.

I'm a big fan of avoiding magic numbers when I program.

So if someone else is reading my code,

I don't want 'em to see eight plus 32 plus 32.

I just want 'em to see variables that make sense to them.

So in Anchor I'm often adding eight to things

to get the true size of an account on chain.

That eight, that eight bytes is the anchor discriminator.

So I'm just gonna do pub const ANCHOR DISCRIMINATOR

and that will be a, what is it? A usize.

So just whatever the size is for the platform

and it's eight bytes.

It just saves us having to reference

that anywhere else as well.

Oh, and I'm missing an equals sign there.

There we go. Great.

We're also gonna be transferring some tokens

in this project.

So let's create a helper function

that just does that for us.

That way we don't have to cut and paste everywhere.

Instructions new file

and we'll call it shared.rs.

And in this file we'll create our transfer tokens function.

So that's just gonna be pub function transfer tokens.

And we're gonna take this generic,

which is the Rust lifetime specifier.

This is telling Rust how to hold

onto a bunch of the variables inside this function.

And what we're saying with this lifetime of info here

is just hold onto them for as long as you need to hold onto

the underlying Solana account info object.

If you dunno what that means, that's fine.

This is not a Rust course, this is a Solana course.

But you can read up about Rust Lifetimes in the Rust book.

Cool. So we're gonna say from,

and that's gonna be a token account.

Token account is fine

if we wanted to make our project specific

to either the old token program,

or the new token extensions program.

But if I wanted to make my program work on both of those,

then there's a thing called interface account,

which is just a way to have our program work across tokens

that are made with the older token program

or the new token program that will last

for as long as info does.

And you can see here how we wrap around our token account.

We're also going to borrow that value as well.

We should add a return type for this function.

So it's just gonna return a result of nothing,

that looks nice.

We should probably also for our Share.RS,

we should actually expose it from our module

'cause I can see that everything is a little bit dark

and gray here.

Let's do that too.

So we've got pub mod,

initialize and pub use initialize.

Let's change that to pub mod shared and pub use shared,

star to expose everything inside shared.rs.

And you can see that the gray has gone away now.

You can also see that interface account is red if...

Yeah, anchor lang or Prelude InterfaceAccount,

that looks good.

Yeah, I think we want that.

In fact we can just import all of the anchor prelude

'cause it's kind of a bit of a normal anchor thing

just to import the entire prelude.

That gives us all the what people often call the goodies

from Anchor without having to type the full path

to any of them.

We'll probably also need to import a bunch of things

from anchor-spl.

So let me do that.

Like token account and we wanna import it

from anchor-spl token interface

because that is the magic that allows our program

to work on both the older token program,

and the newer token extension program.

That is looking a little bit better.

I'll just fix a little typo there.

And I will also turn on word wrapping

so you can see what I am doing.

There we go. Cool.

So it's still a little bit of red there

but it's just saying that we're not returning a result yet,

which is true.

We're not returning a result.

I'll also neaten this up a little bit.

Here we go.

And let's add more of these options

for our transfer tokens function.

So we have a from which is an interface account.

We'll also have a two, which is an interface account.

Also wrapping around token account,

an amount which we will also borrow.

And that will just be a U64,

a mint which will be a pub key.

Same thing we'll do interface account. Oops.

And we will...

It's gonna live for the lifetime of the account info object

and it will be a Mint is the anchor type we're using there.

We'll have an authority

which will be signing this transaction.

So borrow, Signer and that has a lifetime of info.

We'll also have the token program be one of our options

as well.

And the way we say that to say that our token program

can be the older token program or the newer token extensions

is interface and that lasts for info.

And then we say token interface.

There we go. That looks good.

So the way we actually transfer tokens

is we call the token programs transfer checked function.

Transfer checked just does a little bit of logic,

making sure the mints are okay

and a bunch of other useful safety features.

This is the reason we use Anchor 'cause it provides us

with this good stuff right out of the box.

So the first thing we're gonna need to do

is specify all the options we want for transfer checked.

We'll just get them from the options we had

for our argument.

So we'll say let transfer accounts options

and that's a type called TransferChecked,

a struct called TransferChecked.

So in Rust sometimes you have structs

which would be interfaces if you're familiar

with say TypeScript,

that have names that kind of seem like functions.

All that means is that, that struct is an object

that is used as an argument for that function.

So we're about to call a function called transfer checked.

The options for that function are called transfer checked

in title case.

Just a bit of a kind of cultural thing in Rust.

I would probably call this,

if I was writing this in TypeScript I would call this

something like transfer checked options.

But in Rust, it's just transfer checked in title case.

That's the options for the transfer check function.

Because we're calling just a program in raw Solana,

we're not doing any Anchor magic here.

We're gonna turn everything into the low level Solana

account info object.

So you'll see TransferChecked,

from will be the same as from and just to account info.

And mint will also be mint to account info.

You might be surprised to find out

that two is to.account info.

And our authority is also authority.to account info.

We do this once

'cause we don't wanna have to run .to account info

a bunch of times throughout our program.

So that is our options.

So then we should just call it transfer.

Yeah, that looks good.

We can call it transfer accounts options. That's fine.

So our CPI context,

CPI context is just really the program you're using,

plus the options.

That's a CPI context.

New and we have our token program and guess what?

It's to account info and our transfer accounts options.

And that looks good.

We've got a little bit of red here.

I think it just wants us to import Mint,

and we want to import Mint

from the token interface option's.

That one there.

Likewise TransferChecked.

We want to import

that also from the token interface as well.

So token, token 22, just see if that looks right

'cause that should just be TransferChecked.

There we go.

We don't need any special magic there.

anchor-spl token interface, Mint, TokenAccount, token interface TransferChecked.

So there are similar reports in other places.

We wanna make sure we're always importing the ones

from TokenInterface.

So we have our CPI context, which is our program

that we're about to call plus all the arguments.

Let's actually call our program.

So we do transfer checked, and the cpi context,

the amount and the decimals

which we'll get from the mint.

And that looks pretty good.

We just need to import transfer checked.

And again we want to import it,

probably from token interface I think.

So we'll just do it manually.

Trans, sorry, transfer_checked

and that looks good.

No red. Everything is importing

from either anchor lang itself just for the prelude

and everything else is imported

from anchor-spl token interface.

Great. So we now have this reasonable function wherever we need to transfer tokens

and we're gonna call it a whole bunch.

So that is useful.

Let's also think about what we're gonna write

to the blockchain.

As we talked about before, there's really two parts to this.

There's a make offer instruction handler

which will send tokens into a vault and write information

about what the person wants in exchange for those tokens.

And there's a takeoff instruction handler

which will allow somebody to take the tokens from the vault

and also send their tokens directly to the person

that made the offer.

So in state.rs, we're gonna make a little file

that talks about what that offer should look like

and we're gonna call it offer.rs.

And we're gonna import our anchor prelude again,

oops, use anchor lang, ding ding,

prelude, ding ding, star.

Ding ding is just a personal thing I use to refer

to the double colons.

And then we're gonna talk about

what that actual offer looks like.

So that's gonna be a struct pub struct Offer.

This is gonna be an anchor account obviously.

So we add out hash account macro,

also we want to easily be able to work out the size

of what an offer is 'cause we're gonna have to write

one of these out to the chain at some point.

So there's actually a useful thing in Anchor

that's called InitSpace,

which is the size taken to initialize

or create this account on chain.

So if you add derive InitSpace,

our offer struct or any instances of our offer struct

to now have an Init Space property that is added to them,

or an Init Space trait that is added

to them in Rust terminology.

As we said before, this is the offer

where we save details about you know,

who's making the offer and the tokens they want, et cetera.

So it's gonna have an ID

which we'll just use an numeric ID. U64 is fine.

A maker which is who made the offer,

and that will just be a Pubkey.

The mince of what they are providing and what they want.

So we'll call that token mint a and that is a Pubkey,

and I will copy that line down.

Shift option down.

Token mint b is also a Pubkey.

We have token b wanted amount,

which is how much they want in exchange

for what they've put into the vault, which is a U64.

And also 'cause this is a PA, we want a bump

which would just be eight bytes at that missing pub.

And everything is gray because I haven't actually exported

the contents of offer.rs.

So back in state mod.rs I will add pub,

pub mod offer and I will say pub use offer

everything inside.

And you can see now that offer.rs looks a little bit better.

What is that U?

Oh, I've got a semicolon on the end there.

I don't need that.

There we go. Nice.

All the red has gone away.

We are ready to to move on.

So we've just made our offer,

which is what we're gonna write to the chain

and we should probably get started

on our instruction handlers.

So let's open up lib.rs

and just have a look at how it looks.

It is essentially the same

as most other Anchor programs, has a bunch of imports.

We say our programs ID or program address

and then we have this module that we turn

into an Anchor program with the program macro

that, yeah defines all our instruction handlers.

There's one instruction handler called initialize.

Let's turn that into our make offer instruction.

So I'm gonna call it make_offer.

I'm gonna change the spelling of CTX to just be context.

That way I don't have to think

about it being spelled slightly different.

And it's gonna be a context of MakeOffer, we'll call our accounts for MakeOffer,

MakeOffer rather than calling them initialize.

Again, I don't particularly like calling structs things

that like verbs like MakeOffer,

but it's a bit of a Rust convention to...

If a program or a function has some options,

then those options are named after the function

just in title case.

So inside our MakeOffer instruction handler,

I'm gonna call two helper functions.

The first will be instructions make offer

and we'll call that first function send,

offered tokens to vaults

and we add a little question mark on the end

to say if this fails, throw an error.

You know, and that's okay,

we expect that maybe it will fail in some cases.

And our second function will be called instructions:

make offer save offer.

Examples of failures in sending the offer tokens

to the vault was say we tried,

like someone was trying to re-initialize

an old vault account, we would throw an error

in that function.

So that question mark is just a way of saying,

hey, if it throws an error, that can happen.

We expect that.

It's a short way of doing error handling.

So we have send offer tokens to vault and save offer.

That looks good.

We should probably make make offer that sounds good.

I think it's...

Yeah, it's looking for MakeOffer

and it's not finding it so it's not happy with that.

So we need to make the MakeOffer struct

which is all the accounts that people will need people

to specify that they're gonna use when they wanna use

the MakeOffer instruction handler.

And we also need to make the actual MakeOffer file.

So in instructions let's rename our initialize

and let's just call it make_offer.

There we go.

And happy to do any edits for us.

Vs code is just updating module.rs

to specify that file is now called MakeOffer

rather than being called initialize.

Back in make offer.rs what what used

to be called initialized.rs,

we're gonna rename that to MakeOffer

that struct up the top and we will also rename handler,

we'll turn that into our send offer tokens

to vault instruction and we'll work on that in a moment.

Let's just start with MakeOffer.

What kind of accounts will people need

when they're calling this?

Firstly we'll need their own account

'cause they're gonna sign the transaction.

So we'll call that the maker, that's pub maker

and that is a signer similar

to before we're going to use this

info generic which is the Rust lifetime

of an account info object to tell Rust

that the lifetime of these variables

in memory will be the same

as the underlying account info object.

So we have a signer of info

and because the maker will pay for the transaction,

their account needs to be mutable.

So let's do account mute for mutable.

We will need to have the mint of the token

that they're putting into the vault.

So we'll call that pub token mint a,

we're gonna use an interface account again

just to say this could be a mint account

from token program or a mint account from token extensions.

So interface account and the lifetime is the lifetime

of the underlying account info object

and mint is the object we're gonna effectively wrap around.

Will also make sure that the token program used to mint

that is what we expect.

So we'll say mint ding ding token program

equals token_program.

We've done our import,

and now we're gonna have the other mint

which is token mint B.

I'm just gonna copy and paste that one,

and it's gonna have that same check.

That mint token program equals token program.

So that's the type of token that we want, token mint B.

We'll also specify where the tokens that we are about to put

in the vault come from, which is maker_token_account a.

That will be an interface account again.

So wrapping around token account,

and lasting for the lifetime

of an account info object, the underlying.

Rust account info from the Solana crate rather than.

Anchor has nice names like token account,

mint account, et cetera.

Whereas the Solana program just calls everything

an account info object.

We're gonna add a few Anchor checks

to maker token account a.

The first is that it's gonna be mutable

'cause we're gonna move files out, not files,

we're gonna move tokens out of that token account

into the vault.

We'll have a few checks as well.

Associated token mint must be token mint a,

and the associated token authority,

fix that typo, equals the maker.

In other words, this account actually needs to belong

to maker of the person signing the program.

And associated token program

should be equal to token program.

That looks good.

I'm gonna keep going

'cause I've got a little bit of red there,

but I need to finish some more things off

before I look at it.

Pub offer. So this is the offer account

which we'll be making, that's just an account.

It's what we call a system account.

So nothing particularly special,

but we will add some Anchor checks

'cause this is this PDA where we're going to save details

about what we want in exchange for our tokens.

So we'll say init, I.e make it,

who will pay to create it,

which is the maker, I.e the person

who signed this transaction,

how much space we're gonna allocate,

which is eight bytes ANCHOR DISCRIMINATOR.

That's what we made in our Constance IRS before.

Plus the size of offer, INIT SPACE.

So the size for everything to store an offer struct.

Seeds will be the bytes of the word offer.

The key of maker, the pub key of maker

and as a reference, the idea of the offer.

And we need that to be bytes.

So let's turn it into little Indian bites

and then also as ref.

That looks good.

Oh and finally, 'cause this is a PDA

and we wanna be efficient, we will save the bump.

That looks good.

Got a few things we probably need to import.

Let's go and check.

So we've got our anchor lang prelude star,

and we're importing a few things.

Oh wait a sec.

We're importing a few things

from our anchor-spl token interface,

but we probably should be importing a few more.

So I would say actually we want from anchor-spl,

firstly yes we are happy with token interface mint,

but we probably also want to import AssociatedToken.

Thank you very much.

And rather than just mint, we'll also want TokenAccount

and TokenInterface.

That should be good.

A semicolon there.

And then there we go.

Couple of those we're not using yet but that's okay.

So we've added those imports,

and you can see there's a lot less red there.

And again always using token interface

rather than some of the other imports

just so we can have that compatibility

with both the older token program

and the newer token extensions program.

So we have our offer,

we need to also use the Vault account

'cause you know when we make an offer,

we're gonna say details of the offer

and we're gonna put our tokens in a vault account.

So let's do pub vault,

and that's also gonna be an interface account

with a lifetime of info and that will be a TokenAccount

'cause it's just gonna store some tokens for us.

Let's add some anchor goodies as well.

Some little checks on that account.

Firstly we're gonna need to create it so init,

we don't say init if needed

because we don't want anyone to be able

to like reuse old vaults.

We'll say who is gonna create the account,

which is the maker.

And we're gonna mention a few checks

that we're gonna do with this

and I'll copy it from maker token account A.

We're gonna make sure that this token account's mint

is token mint a.

The authority will not be the maker,

it will actually be the offer.

So this token account isn't owned by a user.

It's being owned by our program,

specifically actually it's authority

rather than its program owner is the actual offer account.

So the vault account is owned by the offer account.

The vault's authority will not be maker,

the vault's authority will actually be the offer

because the offer PDA is going to...

Each individual vault will be...

Its authority will be the offer account.

The offer account will sign for things

to move things in and out of the vault.

So you'll see that later on

when we start transferring tokens around.

And we'll also check the token program

is the token program we expect.

Finally we are just gonna specify the accounts

of the programs we use.

So like the system program,

system_program is program account.

We'll use it for the lifetime of info and system.

And likewise, we'll also use the token program,

that will be another interface account

because we're gonna wrap around

either the older token program

or the newer token extensions program.

So that's token program is a interface,

sorry, it's not an interface account,

it's just an interface over info and tokenInterface.

We can actually make our project a little bit smaller

by excluding this stuff, but it is best practice.

So that's why we've included it in this tutorial.

Finally, we have the associated token program,

which is the program that maps associated token accounts

to their owners and to token mints.

So associated token program is a program

for the lifetime of info and associated token.

There we go.

So that looks pretty good.

It's saying it doesn't like ID so let's go fix that.

We need to actually say in the struct

that we want access to the ID

that the client provided

when they were making their instruction.

So we will scroll up and we'll say we want access to,

was it instruction?

We want the ID parameter, which is a U64.

Save that and you can see the ID Green has,

sorry the ID Red has gone away.

ANCHOR DISCRIMINATOR.

It looks like we didn't import that.

So import create ANCHOR DISCRIMINATOR.

Thank you very much. Sorry.

Offer is not an account of system,

is an account of offer.

In other words offer is the struct and in our...

We are specifying that the offer account

is an instance of that offer struct.

So yeah, our whole MakeOffer is complete.

There's no red.

We now need to work on send offered tokens to vault.

So let's make send offered tokens to vault.

This is actually a pretty small function.

It's gonna take a context which I will spell properly.

And so it's gonna take a context of MakeOffer

which we will actually borrow

because we don't need to write to it.

And what else is it gonna take?

Probably the amount of tokens.

So token a offered amount, which is a U64, right?

There you go.

And we're gonna get rid of the logging there.

We're just gonna say transfer_tokens.

Oops, tokens. And we'll say our options for transfer tokens.

So the first thing is contact.

I think I might need to import this actually.

Is it complaining that I need to import this?

No, it's just saying that we...

'Cause this is gonna return a result of nothing, we can't.

There either needs to be a semicolon there and we go okay.

Or we can just remove the okay

'cause transfer tokens returns nothing anyway.

So there we go, and saying that transfer tokens

is not imported so we need to import it

and it looks a little bit happier now.

It is just saying that we need to give it

the right arguments, which is what we're gonna do.

So we're gonna borrow context.accounts.

This is gonna be our from,

it's gonna come from maker token account a

and it's going to the vault.

So context.accounts.vault.

How many tokens?

Well, that is the token a offered amount.

What's next? The mint.

Mint will be the context accounts

and that will be token mint a.

Authority will be the maker.

So that's context.accounts.maker.

And the token program will be whatever

the person running the instruction has specified.

So context.accounts.token program, save it.

And we've filled in all our details

that our transfer tokens functions wants.

There is no red in the file.

We have now created our send offered tokens

to vault instruction and everything in that file looks good.

Let's add the final part of MakeOffer

which is that second function called save offer.

So pub function save_offer.

We're going to...

We'll use the context again.

So it's gonna be a context of MakeOffer

'cause we want access to all those accounts.

So we have our context of MakeOffer.

We're gonna have an ID which is that U64.

We're gonna need to specify

how many tokens we want in exchange

which is token b wanted amount.

And that will also be a U64.

And they are all our options.

So this will return a result of nothing.

And here is our functions body.

I'm just gonna turn on word wraps,

so you can see a little bit easy.

There we go.

There's a little bit of red there.

It's just saying that we haven't returned a result yet, which is true.

So we're gonna call context.accounts.offer

which is the address of the account

that we've calculated based on the seeds.

And all we're gonna do is just use this set inner method

to go and save some stuff into the account at that address.

So we provide an offer instance

that we're gonna make from the ID.

The maker in our offer will be from accounts:

tokens.accounts.maker and .key

which is their Pubkey.

Token mint a will be context.accounts.token mint a.key,

token mint b will be context.accounts.token mint b. Key.

How much of token b they want in exchange,

which is the token b wanted amount.

And also 'cause this is a PDA, we will be saving the bumps

that we used to calculate it just for performance reasons.

Context.bumps.offer and then, oh there's a typo there.

That context.account should be a context.accounts.

And we will also add an okay

'cause we're gonna send this back

to the top level instruction handler.

That looks good.

There is no red in our file.

Very happy with that.

So we've now finished all the functions we need

for MakeOffer as well as the accounts struct

that instruction handler is going to need,

which specifies all the accounts that we're gonna use

in this instruction and how we're going use them.

So we're just gonna go back into this top level lib.rs

file and make sure that our top level instruction handler

has all the things that we want somebody to provide

for this instruction.

So obviously yes they have a context

which includes all the accounts they're gonna use

in this instruction, whether they're readable or changeable

or all the other things that we need,

whether they need to sign the transaction.

But we're also going to need them to specify an ID

which they can come up with themselves.

We're gonna make it so that people can't reuse IDs anyway

so it's a random number that hasn't been used before.

They'll also need to say the amount of token a they want,

token a offered amount which will be a U64

and also a token b wanted amount,

which will be a U64.

That looks nice.

And then in our MakeOffer body,

we need to specify all the things

that these two helper functions are using.

So instructions, make offer, send offered tokens to vault.

You can see from the error that it wants a few things.

It wants a context and it wants the token a offered amount.

So let's give it the context

which is literally just our context from the parent

and we will borrow that

and we want token a offered amount as well.

That looks good.

You can see the red has gone away.

Save offer also wants to use our context.

We can see from the error message it takes a context

and ID and a token b wanted amount.

So context, id and token b wanted amount, save that.

I've got a semicolon on the end of save offer.

I don't actually need it

'cause we're gonna return whatever save offer returned.

So this is just a rusticism, you can emit the semicolon

from a line and that ends up being returned by the function.

That looks good.

Our program should probably build at this point,

we've only done make offer

which is just getting Alice's tokens into the vault

and saving details about what she wants.

That's good enough time as any to go and test our program

by building it.

So let's do a anchor build and build our program.

I'll give it a moment to finish.

And we can see now that we've updated our program

and it builds now, which is great.

So we're now officially halfway there.

Let's go create take offer.

So back in our top level lib.rs,

we're gonna do pub function, take offer

and make a new instruction handler.

That is gonna take a context which is a context of TakeOffer which we will create in a moment

and it's going to return a result of nothing.

And there's gonna be two parts to TakeOffer

which you can probably imagine.

The first is taking the tokens from the vault

and putting those in the taker's account.

And the second is taking the tokens

from the takers token b account and sending them straight

to the maker's token b account.

So let's have a function

for each of those.

Instructions take offer,

this is gonna be a file that we make

and the first function will be called:

send wanted tokens to maker.

And we will pass it the context

and we're gonna borrow the context

because we don't actually need to write to it.

And we're gonna say if that fails, that's fine, it can fail.

There might be many reasons why it fails.

Maybe the person doesn't own

the accounts they're trying to use.

Yeah, that's fine.

If it fails then it fails.

That's anchor just doing its default checks

and saying that something was wrong.

And if it fails that's fine

because it's Anchor probably complaining

that somebody is trying to take an offer that doesn't exist

or that one of our token accounts

isn't actually owned by the taker.

So that's all good.

Instructions take offer,

and we will have withdraw and close vault

and it will take a context

and it will return that.

Now we can see there's a little bit of a red here.

It's just saying that TakeOffer,

the accounts struct hasn't been made, that's okay.

And the TakeOffer file hasn't been made.

So we should probably make our TakeOffer file.

So under programs, swap, source instructions, new file,

take_offer.rs and let's go create takeoffer.rs.

The first thing we're gonna need is,

well let's do our import of all the standard Anchor bits.

We can probably get most of those from make offer.

So I'm just gonna copy all the imports

from make offer into take offer

and we can remove the ones we don't use.

Also in mod.rs I'm also going to make take offer a module

and expose it for everyone else.

So here we go.

So let's create our take offer account struct.

So it's a pub struct of called TakeOffer

and it takes an account info item

to set the lifetimes of various objects inside it,

also 'cause it's gonna be an Anchor accounts struct,

we need to say hash derive accounts

and I need to spell derive correctly.

There we go.

That looks a little bit better

and it's a capital A for accounts as well.

Alright so we have a taker which is the person

who's going to sign this transaction.

That's a signer whose lifetime is info.

And we'll also say because the taker is going to,

no no, not only sign this transaction

but pay for the accounts to be created.

So we'll say account mut, so that we can mutate

or in other words change the balance of this account

so that they can pay for the transaction.

We'll have the maker that they need to specify,

pub maker, choose a SystemAccount,

we'll go for the lifetime of info and that is also mutable

because they will have some of their balances change.

We'll have token mint a and that's an InterfaceAccount

of info and system.

Oops, not system, mint, it's a mint account.

So it's InterfaceAccount wrapping around mint,

that just means that again we can use this

on the older token program,

or the newer token extensions program.

We're gonna specify there's token mint a,

and token mint b as well.

So let's specify both of those.

Token mint a and token mint b.

They look good.

What else will we need?

Well, we'll need the takers token account a

'cause that's where we're gonna put the tokens

that we get from the vault.

So pub taker token account a and that is a box

of InterfaceAccount that lives for info

and wraps around Token Account.

And we'll also have the taker token account B,

which is where we're gonna take tokens from

and send them straight to the maker's token account b.

So I'll just do a quick little copy paste,

I'll add that missing comma.

That looks pretty good.

What's it complaining about there?

Gotta make sure that those are all lined up,

that wraps around.

Yep, it's gone away. Excellent.

Just needed to add another comma.

We're gonna add some anchor constraints

on token account a and b.

We'll say so account and for taker of token account a we'll say init if needed.

Why init if needed?

Because the taker may not have any balance

of token account a.

So if they've never had token a previously,

when we move the tokens from the vault

into the takers token account a,

we'll need to make a token account a for them.

So let's do that, init if needed.

And we'll also say we wanna specify the mint

of this should be the token mint a.

So that is...

Also say who's gonna create this account by the way.

So payer equals taker, it's gonna pay

to make their own token a account.

Init if needed.

Payer is taker.

Associated token mint of this account

should be token mint a.

Associated token authority which is kind of the...

If you think of owners in Solana are like program owners, but authority is like the account level owner.

So that authority is taker

and the token program should just be the token program

that we specified.

So that is associated token.

Token program should be token program.

Cool. So taker of token account a

is a box of InterfaceAccount, TokenAccount.

That looks good.

Init if needed.

Payer is taker and the mint should be token mint a,

the authority should be the taker

and the token program should be the token program specified

in the instruction.

Little bit of red there.

What's it complaining about?

A non-optional in a constraint requires

a non-optional system program field to exist

in the account validation structure.

That's okay. I haven't specified

the system program yet.

So once I add the system program here,

that error will go away.

So we've got the taker token account b

and let's add some constraints here, account and for...

This will be mutable 'cause its balance will change

and all we need to do is copy some of our constraints

from earlier.

But this is taker token account b.

So we need to make sure that we checking

that it is the token mint b not token mint a.

That looks good.

There's another account that will...

Another token account we'll be mentioning here,

which is when we will take tokens

from the takers token account b

and we'll send them to the maker's token account b.

So pub maker token account b,

and let's add some constraints,

so Anchor can check all these good things for us.

The maker's token account b will also be init if needed

because the maker again may not have ever

had any of these tokens before.

So init if needed,

payer to create this account will be the taker

and the mint authority and token program,

I'm gonna copy and just double check.

So the mint for the maker token account b

should be token mint b. That's good.

The authority is not the taker, it'll be the maker

'cause this is the maker's token account.

This is Alice, this is not Bob.

We need to make sure that Alice owns Alice's account

that holds token b.

And the token program is the token program they specified

in the instruction.

That is all good.

Finally. Oh, yes we're gonna involve the offer account.

So the offer is just an account who will live

for the lifetime of info and it's of the type offer.

And what do we need to add to this?

Well, let's start our little Anchor account constraints.

This will be mutable. Why?

Because we're gonna close this offer account

when the offer is taken.

We'll also when we close this account,

we'll also return the SOL init back to the maker. Why?

'Cause the maker made the offer account in the first place.

So it seems fair that when we close this account

we should refund that back to the maker.

Close equals maker.

And we're gonna just do some little...

We're gonna specify some things we want to exist inside

the struct. So has one equals maker,

has one equals token mint a,

has one equals token mint b.

And the seeds for this are exactly the same

as what they were before,

which is the bytes of the word offer

and the maker's key as_ref

and the offer id.

Offer.id.to little Indian bytes.

So we have it as bytes, as ref.

There we go. That looks good.

And finally the bump equals offer.bump.

Always save your bumps.

That looks good. Excellent.

What else do we need?

We need the vault because obviously this take offer

is going to move tokens from the vault

and yes you can probably already imagine

that the vault will be mutable.

So let's go set that up.

The vault is an interface account.

There's the lifetime of info,

and it is wrapping around just a regular old token account.

Again, if we weren't doing this compatibility

between the older token program

and token extensions program,

vault would just be token account.

But because we want this compatibility layer,

the best practice is to use these interface

accounts everywhere.

So we have our account,

and the vault is obviously gonna be mutable

'cause we're gonna take all the tokens out

and we're gonna give them to the taker.

And the rest is just making sure that the mint of the token account is exactly what we expect.

So I'm gonna copy these but I'm also gonna check them.

So the mint of the vault will be token mint a,

because we took tokens from the maker's token account a

and we put them in the vault

and they will be of the mint a.

The authority of the vault account will neither be the maker

or the taker.

The vault account is actually...

It's authority is the offer account.

This means that the vault is controlled by the offer.

It's not controlled by the maker or the taker,

it's controlled by the offer,

which is the account can actually sign for things.

And when we send tokens from the vault,

they will be signed for by the offer's own account.

And our token program will be our token program.

That looks good, save that.

And finally we've got the normal checks

about all our programs.

I'm just gonna copy those from make offer.

System program, token program and associated token program.

Thank you very much.

I will paste those in and that looks good.

No errors, no warnings, just a couple of unused imports.

That's okay.

I'll delete them once I've finished writing this file.

So we've made our take offer struct,

which is gonna specify all the accounts

that we need to check

as part of an incoming take offer instruction.

And we've created the top level instruction handler

in lib.rs, which we called take offer.

And back in lib.rs we said we wanted to have two functions.

One was called send wanted tokens to maker,

and the other was withdrawn and close vaults.

So in take offer.rs, let's create that,

the first of those two functions.

So to make that send wanted tokens to maker function,

we'll do pub function, send wanted tokens to maker

and that's gonna take a context which is a context

of make offer.

And we're gonna borrow that item as well.

Also that's gonna return a result of nothing

and let's make that happen.

So this is actually just gonna be very simple

to send the wanted tokens to the maker,

I.e send the tokens from takers token b account straight

to the maker's token b account.

It's just gonna be transfer tokens,

our little helper function we made earlier.

And the from will be, we'll borrow this context.accounts

taker token account b.

The to will go to the makers token token account b.

Context.accounts.maker token account b.

The amount of token b to send will come from the offer.

So that's context.accounts.offer.token b wanted amount.

The mint will be and context.accounts token mint b.

The authority I,e, the person who controls

that account will be the taker.

And the token program

will be context.accounts.token program, token_program.

That looks good.

And oh and this is a context of TakeOffer.

I did notice that my auto complete

was a little bit funny there, so that explains it.

I think I, yeah I wrote make offer

when I should have written take offer.

So there you go.

So yeah, that actually looks good.

So we are transferring tokens

from the taker's token account b straight

to the makers' token account b.

We're transferring the exact amount that the maker wanted.

We're making sure the mint of the accounts are token mint b,

we're checking that the authority

of the from account is the taker

because this transfer is going straight from the taker

to the maker and we're just making sure

that the token program is the same one that we specified

in the instructions.

That looks good.

So we've just completed our send wanted tokens

to make a function.

The final part for take offer

is to complete the other helper function we made, which is called withdraw and close vault.

So let's make that now.

Pub function, withdraw and close vault

and it's going to return a result of nothing.

The options it will take are just simply

the context again context,

and let's say the context of TakeOffer.

Now we're going to do a transfer here,

but it's not gonna be a transfer from a regular account,

instead we're actually gonna transfer

from our vault account, which is owned by our offer account.

So this is a PDA transfer.

It's a transfer that's being signed for

by our offer account itself.

So we're not gonna reuse our helper function,

which is focused on a different type of transfer.

We're gonna make our own calls to TransferChecked

with a slightly different set of options this time.

So we make the seeds which we will borrow

and the first of the seeds will be the bytes

for the text offer.

You might remember this, this is the seeds we used

to get the address of the offer account.

So we're going to use that

to actually sign this transaction.

The second option, the second seed

for that account will be contact.accounts.maker

to account info.key.as ref.

What else will we need?

The offer ID.

Context.accounts.offer.id.to little Indian bites,

I'll do a little bit of magic on it there.

And then finally bump,

which will borrow context.accounts.offer.bump.

We'll also want to borrow an offer id, so let's.

That looks good.

Little semicolon.

We can actually have multiple signers

when we do a transfer from a PDA like this.

But we only have one in this case.

So we're gonna wrap our seeds

into a bigger signer seeds array.

Let signer underscore seeds equals bigger array and just our seeds.

Do a little magic there. That looks good.

We're gonna make the accounts that are gonna be involved

in our TransferChecked function.

So that is let accounts equals TransferChecked.

This is really the options for our TransferChecked.

From context.accounts.vault.to account info, mint.

Actually let's do the to first.

I think it's a little bit more.

Context.accounts.to is the taker.

These will go to the taker's token account a

because we're gonna take the accounts from the vault

and put them into the taker's token account a.

Because the tokens in the vault are of the type token a.

Taker of token account a, oh yes to account info,mint

which is context.accounts.taker, token, sorry,

token mint a, remain obviously to account info

and our authority, which will be our offer.

Our offer account.

Context.accounts.offer.to account info.

There and.

Doesn't like TransferChecked.

It's okay we just need to import TransferChecked

from, hmm, token 22 or token interface.

I'm pretty sure I want the one from token interface

that'll allow us to use both older and new token programs.

But let me double check.

Yeah that looks a little bit better.

There's actually a TransferChecked

right below token interface.

So make sure sure you import that one,

and not the really long one that I imported.

That is happier now. Excellent.

So we've got our accounts all set up

for this token transfer we're about to do,

we're gonna make a little CPI context.

Let cpi context equals cpi context new with signer,

and all we're really doing here is combining the accounts,

which are effectively the options for TransferChecked,

the token program and the signer seeds which allow us

to sign for this transaction as the offer.

So the program will be:

context.accounts.token program.to account info.

I might actually, you know like the next time

I get some time free I might make a little wrapper for this

'cause I'm getting tired of typing to account info.

Maybe you are too.

And accounts and signer seeds.

That looks good.

Oh wait, we just wanna borrow that

and it will be happy.

Yep, that looks good.

No red. The only red there is our result. That's fine.

We haven't actually written the code for the result yet.

What's next? We've got our CPI context

and let's actually do the transfer now.

So that's TransferChecked, trans.

And takes a context, which is our cpi context.

It takes the amount that we're gonna transfer.

So we're gonna get that from the vault.

So context.accounts.

Accounts.vault.amounts, there we go.

And, oh the decimals.

Context.accounts mint token a. decimals, token mint a decimals.

That's complaining. It wants me probably to import it.

Cannot find function in that scope. That's fine.

Import TransferChecked

and we probably want to import it straight

from token interface.

So not token 22 and not token...

I think we can just import it straight

from token interface manually. Let's do that.

Comma TransferChecked and there, that looks good.

There's a little bit of red elsewhere. What's this?

Check that out in a moment.

Oh I think, oh yeah, that's just it saying

that we need to do our return, which is fine.

We've done our TransferChecked, which is taken the...

That's the withdrawal part of withdrawal and closed vault

that has moved the token a tokens from the vault

into the takers token a account. Great.

If you're wondering why we need to provide the decimals

for TransferChecked, it's part of the checked part

of TransferChecked to includes some safety checks

like making sure that you explicitly provide the decimals.

So we've moved our tokens from the vault

into the takers token a accounts,

the other part of this is to close down the vault itself

so it can't be reused.

So we're gonna make a close account struct,

which is gonna be the options that we provide to another CPI

to the close account function.

So let's do that now.

Let accounts and it will equals close account.

We have three options.

The account which will be the vault itself,

context.accounts.vault,

and to account info we'll have our destination,

which is where we're gonna refund any Lamports

that were in that account.

We will send them to the taker,

but we could send them to the maker as well.

Context.accounts.taker.to account info

and the authority, which is who's allowed to.

Who's doing this.

Which is the offer, the offer account owns the vault.

That's context.accounts.offer

and you guessed it, to account info.

That looks good.

I needed to import CloseAccount by the way as well.

I've done that right at the top here.

And it comes directly from token interface.

So make sure that you import things from token interface,

not from the old token program specifically,

not from token 22 specifically import from token interface.

Great, so here's our accounts.

Let's make a CPI context,

so that we can go and invoke our close account function.

That's a CPIContext and new with signer.

The program will be context.accounts.token program

and to account info.

The accounts we've actually already specified above

and signer seeds.

I think we just need to borrow that and it should be okay.

Yep, that looks fine.

And then so we have our CpiContext,

everything we need to go and invoke close account.

Let's actually invoke a close account.

Close account and yeah, CpiContext.

Oops. What's it saying there?

Mismatch types.

Ooh, I can see what I'm doing there.

I am not using the right version of CloseAccount.

So let's just go back to my imports and double check that.

Yeah, I've got token CloseAccount.

I want token interface CloseAccount.

It's one thing to be really careful

of when you're using an editor that suggests imports.

Do make sure you're importing everything

from token interface, not from the token program

and you'll see that, that's now fixed.

Great. And I actually can omit the semicolon

'cause we're gonna return whatever ClosedAccount

returns from this function.

That looks good.

So we've created our make offer.

We've created our take offer,

the high level instruction handlers

and all the helper functions and the structs of accounts they need.

Let's go and review lib.rs.

We've got our takeoff or withdrawn closed bolt.

That all looks good.

We should probably check that our program compiles.

So back over here in our terminal,

let's do a anchor build.

A couple of unused imports, that's okay.

I might actually get rid of those

just for the sake of cleanliness.

We weren't using ANCHOR DISCRIMINATOR.

Oh, we need to handle any errors in TransferChecked.

So what we can do with that

is we can say a little question mark there and that says,

look, if TransferChecked throws an error, throw an error.

That's, you know, that's okay, we can deal with that.

That looks a little bit better.

So back in our compiler.

Clear the screen and can build again.

And that builds.

We now have a working Swap program

where someone can offer their tokens to be taken and then someone can take that offer,

and receive those tokens directly in their own accounts.

So we've run our program and it builds.

Next step is to test it.

We've already got some tests we've prepared for you

but I'll go through them here.

This is swap.ts under the test directory

and it's pretty simple.

We import a few common bits and pieces, anchor itself,

the chai assertion library.

The Solana helpers make a few variables

and we have one block of tests called a Swap.

That connects to anchor, which allows anchor to use the keys

and the networks set up in our anchor.toml.

And we set up some variables we're gonna use in a moment

for Alice, Bob token mint a and token mint b.

Our beforehand handler is really simple,

it just creates some users, mints and some token accounts

and uses those mints to mint a particular balance of tokens.

So that Alice has a lot of token a

and that Bob has a lot of token b.

There's two tests and the first is our make token test.

Make sure that it puts the tokens Alice offers

into the vault when Alice makes an offer.

So we're simply going to get a list of accounts together

and then send that transaction off,

program methods.makeOffer

with our options the accounts we want.

And our signer is Alice.

Afterwards we're gonna check the balance of the vault

and we're gonna check the offer account has the data

that we expect.

Second test is pretty simple,

it's program methods.takeOffer,

confirms the transaction and checks Bob's accounts

after running the test.

Also we could add a test to make sure

that the vault is closed.

I don't think we are here,

but I can add one in a moment.

You'll also note I run .slow

and then I have a variable set to, I think it's 60 seconds.

Because Anchor tests connect to our local validator.

By default, Anchor shows them as slow,

because Mocha actually shows them as slow

because they take anything that takes more than,

I think half of 30 seconds is shown as slow.

So if I set my slow threshold to 60 seconds,

anything that takes half of 60 seconds

will be shown as slow.

Our test only take 15 seconds

so they won't be shown as slow.

You didn't really have to worry about that too much.

It's just a me thing.

I hate my tests being shown as slow.

I like to establish bit of a baseline

for how long a test should take.

So yeah, pretty simple.

We just have one described block and a test that goes

and checks make offer,

and another test that checks take offer.

If we run those and you can import them

into your own program, you can get them from

the link that we will add to this presentation, you can actually call your take offer and make offer

and check that they work as expected.

Give it a sec.

As I said, these tests take about, I don't know,

maybe 15, 16 seconds each.

Then we can see the first test is finished,

which is puts the tokens Alice offers into the vault

when Alice makes an offer.

And in a moment our second test

should hopefully finish as well.

There we go.

Puts the tokens from the vault into Bob's account

and gives Alice Bob's tokens when Bob takes the offer.

So this program is a great example of the benefits of Defi.

Alice had some of one token and wanted another token.

She didn't know Bob.

Bob at some point in the future came along,

took that offer that Alice made,

exchanged his tokens for Alice's tokens,

and each side got a 100%

of what the other person was offering without anyone

in between taking a portion of anything.

Also, the other useful thing about this program

is that it shows you how to handle tokens

in your own program, to how to hold tokens

and how to sign to transfer those tokens out of places

where they're stored using CPIs.

So I'll see you in the next lesson.

- Imagine a company that wants to reward its employees

with tokens, ensuring they remain motivated

and committed to the projects.

This process is known as token vesting.

Token vesting is a mechanism used by cryptocurrency projects

to distribute tokens over time

to certain stakeholders like employees,

advisors, or investors.

Instead of granting all the tokens immediately,

vesting restricts access to the tokens over a set schedule.

This aligns incentives and prevents stakeholders

from selling all their holdings at once.

Potentially destabilizing the tokens value.

In the upcoming tutorial,

we'll write our own vesting contract.

Our contract will have the following user experience.

A company will be able to come to the dApp

to initialize their own vesting contract and specify

the mint address and total supply of the lock tokens.

Next, the company will be able to add employees

with their individual vesting details,

which includes the total amount of tokens allocated

to the employee, the vesting duration, and the cliff.

The cliff is the initial period

where no tokens at all can be claimed.

Say the cliff is six months, then for the first six months,

the employee will have zero unlocked tokens.

After the cliff expires, unlocking begins according

to the monthly schedule.

Employees can also use this dApp to view the status

of their vested tokens and claim any unlocked tokens.

This project enables transparent automated vesting

on Solana.

Companies can set customized vesting schedules

for any SPL tokens and employees can clearly track

their vesting progress.

So in summary, token vesting aligns incentives for projects distributing tokens over time.

Our smart contract brings secure

on chain token vesting to Solana.

By the end of this tutorial,

we'll accomplish writing the smart contract code,

running test, creating a front end and connecting

the smart contract to the front end,

creating a fully functioning decentralized application.

Let's get started.

All right, so let's create the vesting dApp.

We're going to be using the MPX Create Solana dApp Scaffold

because it connects both a front end

and a backend in one workspace and makes it a lot easier

when you're making a full decentralized application.

So let's get started by setting up the scaffold.

We're going to navigate to our terminal

and just type in npx create Solana dApp.

And now we're gonna enter the project name.

We'll do token vesting.

We will select the preset for nextjs tailwind,

and then the anchor counter program.

And this is creating a new workspace with NPM.

So it's gonna take a second to load.

We'll wait for everything to generate,

and then we're going to open this workspace in VS code

and get started.

Okay, so this creates a new folder inside the directory

that you ran the command.

So we're just going to cd into that folder

and then we're gonna open in VS code.

Okay. And just to get everyone familiar with this scaffold,

how it sets up the workspace for you,

you have your web folder that has everything

for the front end of the application

and you can see the app, the components,

and all of the needed configurations.

We're going to work with this later.

So we're gonna start out with our Anchor Smart contracts.

So in the anchor folder you can see it has everything set up

for the backend of the application,

we have our programs, and then you go into source

and this is where the main smart contract lives

in your lib.rs.

Now we also have a few other folders generated

that we're going to use later on in this application,

but let's just first start with our lib.rs.

So you can see this is just a basic counter

that was generated.

If you actually run this with npm run dev,

it's going to actually show the front end

and a working counter.

But we're going to change all of this

for our vesting contract.

So what we can do is just delete everything

within the program.

As you now know by working through the bootcamp,

this program macro defines all of the instructions

for your smart contracts.

And then you have the corresponding data structures

for all of your context.

So we're just gonna delete everything related to the counter

and just have the basic setup for an Anchor smart contract.

So that's using the Anchor laying import,

it has your declare ID with your program ID of the smart contracts.

And then this is where all of your instructions

are going to live.

Now what we need to do is rename everything

from counter over to vesting.

So let's start by updating the name of this program

to vesting, and then we're going to navigate

into our cargo file.

So the package name is counter still,

so we'll update this to Vesting.

Okay. And then, let's see,

we're just going to command Shift F

and see everywhere else that Counter is written

and we just wanna update that.

So it's gonna be in quite a few places.

This will take just a second.

Anything related to the TypeScript files for our web app,

we're just going to deal with later

when we start working on the front end.

So all of this we can just ignore for now.

So now we have to set up our cargo.toml for this program.

And in our vesting contract,

we're going to be moving SPL tokens around

as well as using a Solana program.

So we're going to update our dependencies in our cargo.toml

to use both of those grids.

So we can just open up our terminal

and run cargo add.

And we're going to add first our anchor-spl.

Oh, sorry.

We actually need to navigate into our Anchor directory.

So one thing to note whenever you're using

the Create Solana dApp Scaffold is everything related

to the Anchor Smart Contract lives in the Anchor folder.

So whenever you're running Anchor commands

or any of your cargo commands,

it has to be done in the Anchor folder.

So we'll see CD into Anchor.

And now we can run cargo add anchor-spl.

Okay, so you can see our dependency updated with anchor-spl.

And now if you're following along,

depending on the time that you're following along,

this might have a different version.

So if you want to make sure that everything stays consistent

with the rest of this tutorial

and make sure all the dependencies are compatible,

I would recommend using the dependency versions

that we have here in our cargo.toml.

So now we'll also add in the Solana program.

So we'll do a cargo add solana program,

and if you wanna specify a specific version,

we'll just do that at for the version that we want.

So we're gonna use 1.18.17.

Okay, and now our Solana program is updated.

And now another thing that we're going to need

in our dependencies is the init if needed feature

for Anchor link.

So I'll explain what this does later on

when we're actually using it,

but for now, let's just get our cargo set up.

We're gonna do cargo add anchor link,

and add the feature flag.

And this will just be a init if needed.

Okay. And you can see that the dependency was updated

with the feature init if needed.

And one last thing to update in our cargo.toml

is our IDL build feature.

So because we're using the anchor-spl create,

we also need to update our IDL build

to work with anchor-spl.

So in our IDL build,

we'll just add anchor-spl-IDL build.

Now our cargo is up to date,

and we can navigate back to our lib.rs.

So let's think about what we're building here.

We're doing a vesting contract,

and as we explained in the introduction

is we have two perspectives.

We have our employee perspective

and our employer perspective.

So first we're going to think about the employer.

We're going to be creating a vesting contract,

so we have to initialize that.

And then we also have to add in the ability

for the employer to add employees.

And then lastly, we need to allow for the employees to be able to claim their tokens once they have vested.

So let's start with initializing a vesting contract.

So we can first define the instruction

within our program macro, and we'll just do pub fn

and we'll name it, create vesting account.

Whenever you're defining an instruction,

you always have the first parameter

is going to be your context.

So we're going to just define the context.

And then we're also going to want to create a custom data

structure for this context.

So we'll name that data structure, create vesting account,

and it will return a result.

And then we'll write the logic for our instruction.

But first we need to define what this data structure is.

So outside of our program macro, we can go down here

and we'll use derive accounts.

And we're going to just define the struct.

So pub struct, and this will be create vesting account.

And we need to make sure we use the lifetime specifier.

So what this context is doing is it is passing

through all of the accounts that are needed to process the instruction that you're writing.

So let's explain all the accounts that we need.

First, we need the signer.

So we'll write an account and we need that to be mutable

because the signer is going to be paying rent.

So their account state is gonna be changing

when their balance of SOL changes.

So we will name them signer,

and then the type is going to be a signer

with the lifetime specifier.

Okay, so one thing to note

when we're writing all of our accounts is that with Anchor,

each account used in a transaction,

it's declared in a context structure with specific traits.

And these trades dictate

how the account can be accessed like mutability,

which is why we needed to specify the mutability

for the signer because their account state

is going to be changing.

And in Rust, as you know,

all variables are immutable by default.

So here when we define our account macro,

we're going to have to specify all the constraints

for the next account that we're writing.

And we're going to be defining our vesting account,

which is what we're creating with the instruction.

So because we're creating the vesting account,

we want to use the init constraint to specify

that we're initializing an account

when we're writing the instruction.

So let's use our account macro,

and then inside we're going to first specify init.

Now whenever you're initializing an account,

it's going to be taking up space on chain

and that space you need to pay for in rent.

So we need to be able to calculate the space.

So we can just define the space constraint,

And we'll do the ANCHOR DISCRIMINATOR, which is eight,

and then we have to be able to calculate how much space

that it's going to take up.

And that's going to be defined

by what you're actually putting into this account construct.

So one thing that we do need to specify

is what is this vesting account going to be?

This is a custom made account.

So we're going to pause from writing this data structure,

and we'll just go down to define what our vesting account

is actually gonna be.

So on Solana, all smart contracts are stateless,

every state is stored within accounts.

So here we're creating a vesting account

that is going to store all of the information

for a specific vesting smart contract.

So what do we wanna store?

We're going to first define that it's an account.

We're gonna use the account macro from Anchor.

We're going to name this vesting account.

Okay. So this is a vesting account for an employer

for all of their employees at a company.

So what do we wanna save here?

First we wanna know who the owner is.

So that's going to be whoever has special permissions

to update the vesting account.

So we'll save the owner,

and that'll just be the public key of the owner.

And now we're going to be distributing SBL tokens.

So we want to save what the mint is for the SPL token.

So that will also be a public key.

And now this is just an account that's storing state.

We're also going to need an account

that's storing the SPL tokens.

It'll be a token account.

So we wanna be able to store what the public key

of that token account is within our vesting account state.

So let's just do public key, sorry,

we'll just do pub treasury token account.

This is just going to be the treasury account

of an employer's tokens that are all going to be allocated

to employees that are working and receiving vested tokens.

So this will be the treasury token account.

The type will be public key.

Okay, another thing that we'll wanna store

just for referencing, maybe creating a PDA in the future

is the company name.

So we'll write company name, and that'll just be a string.

That can be a user input value

when they're creating their new vesting account.

And then we also wanna save the bump,

and we'll save the bump for both the treasury token account

as well as the vesting account.

So we'll do bump for treasury,

and that's always a U8 type.

And then just the bump for the vesting account,

which is also a U8 type.

So now we have the account,

but how are we going to actually calculate the space?

Anchor actually has a way to do that.

So this account, macro is an attribute applied

to the data structure to designate that it's Solana account.

And then we can also add the derive macro,

and it's an implementation onto this account data structure.

And we can specify that we wanna use InitSpace.

So we'll just do derive and then InitSpace.

And what this does is Anchor automatically calculates

the space for you.

Now you can see that when I wrote that we have an error

that was generated, and that's because the company name is a string value and a string can be extremely long.

So we're going to wanna designate a max length

for what this string can be.

So I can just go in here and actually write max length,

and I'm just going to say 50. Seems good for a name.

Now that error went away

and now we have our account fully defined.

So now that we have that, when we're calculating the space

in our create vesting account struct,

we can just use the InitSpace.

So that's going to look like just the name of the struct

and then InitSpace.

Okay, so now that we have the space calculated,

we need someone that's gonna pay the rent.

So we just have to define who the payer is,

and that's gonna be the signer of the instruction.

And now we're creating a new account.

So since we wanna be able to derive the PDA in the future,

we need some type of attribute that differentiates

the different PDAs for a specific signer.

So here we're just going to use the company name.

So we can do seeds and we'll use the company name as ref.

Okay. And then lastly, because we have the seeds,

we need the bump.

And then we're going to just need to define

what this account is.

So we'll name it vesting account,

and it will be an account with the lifetime specifier.

And it's going to be using the vesting account struct

that we defined right here.

Now, there is an error, and the reason for that

is because we're passing through a company name

and this struct doesn't know what it is.

So yes, we need to update the input parameters

from the instruction first,

and just say we're allowing on the front end someone

to type in what the company name is.

That'll be of type string,

but this still isn't passed through down here.

So to be able to pass this information

through our context data structure,

we're just going to add this macro

that is basically saying we're pulling this variable from the instruction.

So we'll do the instruction and then name the variable.

So we're using the company name of type string.

Okay, the next thing we're gonna pass through

is just the mint account, and that's going to be

what the SBL token mint is for the vested token

for this specific vesting account.

And we're going to be using interface accounts here,

And it will be a mint type.

So we need to import this mint,

and this is coming from our SPL token create.

So you can see here anchor spl-token mint,

but actually we're going to be using the token interface

rather than the token.

So we'll go down to anchor spl-token interface mint

and import that.

So you can see the import updated.

And you can see the vesting account still has an error,

and we'll get to that in a second.

But basically whenever you initialize a new account

on chain, you do need to pass through the system program,

but we'll pass that through towards the end.

So so far in our contact data structure,

we're passing through the signer

who is signing the instruction.

We're passing through the vesting account

that we're creating to hold all of the state

for the new vesting account.

We're passing through the mint of the spl-token

that we're vesting.

Now, we also need the treasury token account

that's going to store all of the tokens

that are going to be handed out to employees.

So we can initialize that at the time

of running this instruction.

So we're once again going to use the init constraint

on the account.

And now here it's gonna be a little bit different

because it's a token account.

So we're going to need all of the information

for the token.

We can do that by running the token constraint.

And we'll define what the mint is for the token.

And that mint is going to be the mint

that we just passed through above on the previous account.

Okay. And now we also need the authority for the token,

and that authority is going to be the token account itself.

So in the future, when we're actually transferring

the tokens, the token account has the ability to transfer

to the employees.

So we'll set the authority to the treasury token account

which is what we're going to name this account.

Okay. And now we just need the payer,

which will be the signer once again.

And then here we're going to define the seeds.

This is going to be a token account,

not an associated token account.

This is because it's going to be a token account

specified just for this vesting contract.

So we wanna create seeds for the PDA

that will make it easy to derive.

So we'll make the seeds.

We'll make the seeds state that it's the vesting treasury,

and we can do that by just typing out vesting treasury.

And then we'll also wanna add in the company name as ref.

Okay, a little typo.

Let's fix that. There we go.

And now, because you specified the seeds,

we also have to add the bump.

And now all the constraints are defined.

So we just want to state what this account is,

and that will be our treasury token account.

And now we're going to update this to an interface account.

And then the type will be a token account.

So now we have our treasury token account,

our vesting account, we have the signer, we have the mint.

So all that's left is two things,

our assistant program and then the token program,

because we're creating a token account.

And this token program is actually going to be an interface,

and we're going to use token interface.

And now you can see we have one more error left,

and that is we need to import token account,

so we'll just import that.

And that's gonna come once again from the token interface.

Okay. And because we're using a string here with vesting treasury, we're going to update this

instead of as ref, it's going to be as bytes.

And that's everything for all of the accounts

that we need for the first instruction

to create a new vesting account.

So now we can write the logic out

for creating a vesting account.

And you can see here there's a warning generated

for a company name, and that's just because we haven't used

it yet in the logic.

And Rust always wants to warn you

if you're passing through a parameter

and it's not being used.

So the first thing we're going to do

is just load in the vesting account,

and we wanna be able to update the state.

So because the context holds all of the accounts

for this instruction, all we can do is just load

in the context.accounts and we can point

to the vesting account.

Now, what we need to do is we need to dereference this,

so we can add this asterisk,

and that is a dereference operator in Rust.

And what that does is it de references the account

that it's pointing to, which is the vesting account

in our case, and it allows you to modify the account.

You need to do this because it's an account reference.

You need to dereference it to be able to modify it.

And this tells Rust that you wanna work with the actual data

and not just the reference.

So now you can update values of the actual data

of the vesting account.

Now we'll just pull in the vesting account type.

Okay. And then we can just update all of the fields

within vesting account to save the information

for the new account that we're creating.

So first we'll do owner,

and that will just be the key of the signer.

So we can pull in the signing account

and then just grab the key.

Now we need the mint,

and we'll just pull in the mint account and grab the key.

Now we need the treasury token account.

Once again, we'll pull in the treasury token account,

grab the key, and then the company name

is just going to be the company name that's passed through.

And then we'll just grab the bump of both the treasury

and the vesting account.

Okay. Sometimes Copilot tries to auto-fill things,

and it's not always correct.

So we're going to fix what Copilot just wrote.

And we're actually gonna pull this in by doing ctx.bumps.

And we're going to pull the bump

of the treasury token account.

Okay. And then we need the bump

for the vesting account.

So ctx.bumps.vesting account, okay?

And I spelled account wrong, so we'll just fix that typo.

Here we go.

And then we're just returning an okay for a result.

And that's everything we need for our first instruction.

So you can see that most of the logic here happened

within our create vesting account data structure,

because that's where all the accounts are handled.

So if we go back down to our create vesting account, you can see we've set up both our vesting accounts,

and a token account for the treasury.

And then the logic within the instruction

is just saving all the information that we need

to the account state of the vesting account.

So first instruction is done,

now I can move on to the next.

So not only do employers need to have a vesting account,

but employees need to have a vesting account as well

and that's gonna hold different information.

It's gonna hold all the information for their vested tokens,

like the cliff time, how many tokens they have allocated

to their public key, and anything associated

with their token vesting.

So that's going to be input by the employer

of what their specific allocation is,

and then the employee needs to be able to access that

to be able to claim their tokens.

So let's create the employee vesting account.

Now because we're creating a new account,

let's define what is being saved at the state.

We're gonna do the same thing that we did

with our vesting account by defining what the account is.

So we'll use the account attribute

that just tells that this is a Solana account.

And then we're once again going to use derive InitSpace

because we need to calculate

how much space it's gonna take up on chain.

Okay. And then we'll just define what this struct is.

So this will be the employee account.

So what do we wanna save here?

We wanna know who's gonna be able to claim this.

So let's define what the beneficiary is,

and that'll just be the public key of the employee

for this specific account.

So typically when you create a vesting account,

you wanna keep track of when the allocation started,

when it's going to end, and what the cliff time is.

So let's define all of that first.

So the start time.

Now this is all going to be stored in Unix time,

which is an I64 type.

The end time, also an I64, and then the cliff time.

And what the cliff time is, is just how much an employee

has to wait before any of their tokens unlock.

And then once the cliff passes,

it typically unlocks literally after.

Now, we'll also wanna keep track of the vesting account,

and this is just the corresponding vesting account

that was just created by their employer.

So that will be a public key

because it will be the account address.

And now we just need to know

how many tokens are being allocated to the employee.

So we'll do total amount, and that will be a U64 type.

And then we wanna keep track

of how many the employee has withdrawn from their account.

So we're able to calculate at a specific time

how much they're able to either claim,

or how much they still have left to claim in the future.

So total withdrawn, and that'll be a U64 type as well.

Okay, so that's everything we're gonna be storing

to the state for the employee account.

Now we can write the instruction to create this.

So pub fn, and now we're going to create employee account.

Okay. So we'll pass through the context

of our first parameter, as always,

when writing an Anchor instruction

to find as a context type.

And then we're going to create a data structure

for create employee account.

So let's pause from this instruction and go down here

to write our derive accounts data structure.

So pub struct to CreateEmployeeAccount,

add in our lifetime specifier.

So now we want to know who the owner

of this employee account is going to be.

So the owner is actually going to be not the employee,

but the employer because the employer

is going to be the person who has access to update

the allocation information for a specific employee.

So that's why we're going to update,

the signer is going to be the owner.

And once again, it needs to be mutable

because we're creating an account.

So the signer has to pay rent for that account.

So we're going to set the signer as the owner.

Okay. Now we also need to know who the beneficiary is,

which is the employee that's gonna be able

to claim tokens later on.

So let's pass through the beneficiary account

and that'll be a SystemAccount,

and we'll add in the lifetime specifier.

Okay. Now we need to pass through the vesting account,

and that is because we're going to need all

of the information from the vesting account state.

But one thing we wanna do is we wanna use

the has one constraint to just make sure

that the owner of the vesting account is the signer

of this instruction.

And that's just validating that the person

with the correct access is going to be able

to run this instruction.

So we'll do the account attribute,

and add in the has one constraint

and set that to owner.

Okay. So now we have to actually create

the employee account.

So we're going to use, once again,

the init constraint on the account.

So account, init and this is just an account

that is holding state, so we don't need to use

the token constraint.

And we're going to specify what the space is.

We'll use the eight discriminator from Anchor,

and then we can just use InitSpace to calculate the rest.

So EmployeeAccount InitSpace,

and then because we're taking up space, we need to pay rent,

so we need to define who the payer is.

And then we're going to define the seeds for our PDA.

So here, let's use employee vesting.

And then we're going to add in the beneficiary key as ref,

and then the vesting account key as ref.

And now that we have c's, we also need a bump.

And then all that to lift is just to name

what this account is.

So it'll be our employee account. Okay.

And now we have an error here,

and that is because we need to define the system program whenever we're initializing a new account on Chain.

So we'll pass that through, and now our error went away,

and that's everything we need for an employee account.

So we can go back to the instruction

and finish the logic there.

Okay, so we'll load in the account the same way,

we need to dereference,

call the contact accounts and employee account this time.

We'll use the EmployeeAccount struct.

Okay. And now we need the beneficiary as the first field,

and that'll just be the context accounts beneficiary key,

and we have a typo.

It's accounts with an S. There you go.

And now we need the start time and the end time,

but we don't know what that is.

So we wanna update our parameters to include start time

and end time, and this will be passed on the front end.

Okay. So we have our start time, we have our end time.

Now we also need to pass through the total amount.

So let's add that, that's a U64 type.

Okay. And then the total withdrawn

since we're initializing a new account here,

it's impossible to have withdrawn anything.

So we'll just set that to zero.

And then we need the cliff time,

we'll pass through as a parameter.

And now we need the vesting account,

and we'll just pull in the accounts vesting account key.

And lastly, let's save the bump.

Now this will be the bump of the employee account.

So we can just do ctx.bumps employee account.

Okay. And now we just need to return okay.

So now we have an error,

and it's saying there's no such field as bump,

which means we forgot to define the bump

in our EmployeeAccount state.

So we'll just go here and add that field

because we always wanna save the bump.

There we go.

So everything's good

for creating an employee vesting account.

Now the last instruction that we need here is allowing

the employee to come and claim tokens.

So this will have a little bit more logic to make sure

that an employee can claim tokens at the correct time.

So let's start by defining this instruction.

We'll do pub fn claim tokens.

Okay. So we'll pass through the context

as the first parameter, and then we'll go down

and define what that is.

So claim tokens.

And now we can do derive accounts for claim tokens.

So this will pass through all of the accounts

that are needed for an employee to be able to come

to this program and claim tokens.

So first we need the signer of the instruction.

And in this case, instead of it being the owner

of the vesting account, we want it to be the beneficiary

of the employee account.

So we'll update this to account,

it needs to be mutable and it will be the beneficiary,

and that will be the signer of the instruction.

And next, we need the employee accounts.

So we can define the account.

And here we're going to need to update information

in the account state

because we'll be updating total withdrawn

if an employee withdraws tokens.

So we wanna make the account mutable,

and now we need the seeds.

So this will be the same seeds from the EmployeeAccount.

So just to make things easier

and make sure we have all the correct information,

we'll go back to when we initialized the EmployeeAccount

and just copy those seeds over.

Okay. So this is employee vesting, the beneficiary

and the vesting account.

And now we need the bump.

And because we saved the bump when we were initializing,

we're able to just pull that information,

so we can do employee account.bump.

Now we're gonna need to do two checks on this account.

We wanna make sure that it has the correct beneficiary,

so only the employee that is the beneficiary

of this employee account is able to come in and run

the instruction claim tokens.

And then we also wanna make sure that this is connected

to the correct vesting account that's being passed through.

So we'll do has one beneficiary,

and we'll also do has one for vesting account.

Okay, so we'll just define this as the employee account.

Now there's a few things here that we haven't defined yet.

So the beneficiary was already passed through as the signer,

but we need to define the vesting account.

And the vesting account is used twice,

so we wanna just make sure that, that account

is passed through so it's able to pull

the correct information.

So let's pass through vesting account next.

And we need the seeds for the vesting account,

and we'll just do the same thing.

We'll go back to when we initialize the vesting account

and copy over the seeds.

And then because we saved the bump

to the vesting account state,

we can just grab that information.

And then we're gonna do two checks here as well.

So we wanna make sure that this vesting account

has the correct token account that's being passed through,

and that is the treasury token account.

So when this claim happens,

it's going to be transferring SPL tokens

from the treasury accounts into the employee token account.

So we need to pass through the correct treasury account

to be able to pull those tokens from.

So we'll do has one and the treasury token account.

And then we also wanna make sure it has the correct mint

address of the SPL tokens.

So we'll do has one mint.

So then we'll just define this account,

and that's the vesting account.

Now, there's a few things that we haven't defined yet,

so one is the company name.

Now we're gonna have to pass through the company name

as a parameter from the instruction.

So let's go back up to the instruction,

and let's add in company name here.

And then for this data structure to be able to recognize

the information we have to pass through our instruction with the company name.

Okay. So now you can see that error went away.

Now we need both the treasury token account

and the mint account to be passed through.

So let's do the mint next,

and that will be an InterfaceAccounts mint.

Now we need the treasury token account.

So this also has to be mutable

because we're going to be changing

the token account balance.

And all we have to do is just define what this is.

So treasury token account,

it's an InterfaceAccount token account. Okay.

And you can see both of those errors went away,

and all the accounts are passed through.

Now what else do we need?

So we have the beneficiary, we have the employee account,

we have the vesting accounts, we have the mint,

and we have the treasury token account.

So it seems like there's one important thing that's missing,

and that is the employee token account.

So we have the account state for the employee,

but we don't have where the tokens are gonna be sent to.

So what we can do is create an associated token account,

and this is where the init if needed constraint

is gonna come into play.

So let's first define the account

And instead of init, we're going to use init if needed.

And that's because some users may already have

their token account for the specified SPL token created.

And if they do, it's going to just bypass

the init constraint and continue with the rest of the logic

for the instruction.

But if they don't already have this initialized,

this account will then become initialized

after running the instruction.

Now we'll need the payer because it has init if needed.

So if the user doesn't already have

this account initialized, then we need a payer

to be able to pay rent for the account.

Okay. So in this case, the payer's going to be

the beneficiary because it's their associated token account.

Now we're going to just define

what the associated token account is.

So we'll use the associated token constraints.

And first we want the mint,

and that'll just be the mint address of the mint account

that we're passing through.

And then we also want the authority,

and that'll be the beneficiary

because it's their own token account.

And then we also need to define the token program.

So associated token, token program,

and that will just be the token program.

And now all that's left is to define what this account is,

and this will be our employee token account.

And because we're using the token interface,

this will be an InterfaceAccount token account. Okay.

Now, one thing is missing is what is the token program.

So let's define that.

And the token program is an Interface, TokenInterface.

And now because we're using the associated token constraint,

we also wanna pass through the associated token program,

and this is associated token.

And then lastly,

because we're potentially initializing a new account,

we wanna pass through the assistant program. Okay.

And now that error went away,

and this should be all of the accounts that are needed

for claiming a token.

But let's just double check.

So we have our beneficiary which is the signer,

and we wanna make sure that the beneficiary is the signer

because we only want the beneficiary

of the specified employee vesting

to be able to claim tokens.

And then we have the employee account,

and that is just all of the state

of this employee's vested token information.

We're doing a check to make sure the correct beneficiary

is signing, and also it's connected

to the correct vesting account to grab the information

for the mint and the company.

Now we have the vesting account being passed through,

we're checking the treasury token account

is passed through correctly, and that's needed to be able to pull tokens from to send to the employee.

And then we have the mint,

which is to know what spl-token we're sending.

Now we're passing through the mint account,

we're passing through the treasury token account,

and then we're creating if needed an employee token account.

And that's the token account that's going to claim

the spl-token for the vested token.

And then just the token program, associated token program

and system program need to be passed through.

Okay. So that looks like everything.

Now we can go back to the logic needed to claim tokens.

We can return the result.

And I like to always, before writing all of the logic,

just have our okay return value at the end,

so Rust doesn't get angry with us

and have a bunch of errors as we're writing.

First, we need to load in the employee account

to get all of the information for the vested token.

So let's load in the employee account.

And here we're going to do a mutable reference.

That is because we need to specify

that the referenced account is going to change information.

And let's do ctx accounts.employee account. Okay.

So that loads in our employee account.

It's being stated that it's a mutable reference.

Now, earlier when we loaded accounts in,

we used the D reference operator,

but here we're using mutable reference.

So let's talk about the difference between the two

as we're writing this out.

So D referencing is used when you need to manipulate

the actual data stored in the reference location,

and it's kind of less common at a high level operations.

And it's mainly used

when you're initializing your resetting account data.

Earlier we were initializing two accounts,

so we were using the dereference operator,

but here we're borrowing mutable reference,

and this allows you to pass employee accounts around

in your function or to other functions while still having

the ability to modify the original data.

And this is useful when you're using multiple operations on the data, which we're going to be doing

as we're calculating to be able to claim tokens.

When you're claiming tokens, the first thing you wanna know

is the current point in time.

So we can know if an employee

is able to claim tokens or not.

So let's first use the clock to figure out what the time is.

So we'll do now is equal to Clock.get and we'll unwrap,

and we need the Unix timestamp.

Okay. Now I wanna check if the current time

is before the cliff time

because if it is before the cliff time,

then no tokens can be claimed.

So if now is less than the employee account.cliff time,

we wanna return an error.

And now we're gonna create a customer error

for this program,

and we'll name it ClaimNotAvailableYet.

And it's always important to define custom errors

when you can because it allows a user

to understand more of why they're getting an error when they're interacting with the smart contract.

So we can define custom errors

by using the error code macro from Anchor.

So we'll just do error code.

So this error code attribute, it's applied to the enum

that it's attached to, to generate a result type T

or an error type.

So that's used to return errors for the program.

Okay. And we're using an enum here

because you're only gonna be calling one error at a time.

Okay. So we'll do claim, oops,

we'll copy in the error

that we specified just to make sure everything matches.

Okay. And we can add a message

that's gonna be displayed for the user.

So we'll do message: ClaimNotAvailableYet. There we go.

And now you have a customer error in your program.

Okay. So next we're going to have

to start doing some calculations,

but when you are running calculations in Rust,

we have to add some additional checks.

So we need to take into consideration both underflow and overflow errors.

Underflow occurs when calculations result in a number

that's smaller than the minimum value

that can be stored in the data type.

And overflow occurs in the reverse

when a calculation results in a number that's larger

than the maximum value that can be stored in the data type.

So we're going to use specific methods

when we're writing out these calculations.

So let's first calculate the amount of tokens

that are able to be claimed.

First, we're gonna define the time since the start

of the tokens being vested.

So we'll do time since start and we'll do now.

And instead of just subtracting,

we're going to do a saturated sub.

And what that does, it just ensures

that the subtraction doesn't go below zero,

which can help prevent underflow errors.

So we'll do .saturated, saturating sub,

and we'll be subtracting the employee account start time.

And this is an I64 type because it's in Unix time.

And now we need to find the total vesting time.

So we'll do let total vesting time,

and this will be the end time minus the start time,

which gives you the total time period

that the tokens are vested for the employee.

So employee accounts.end time,

and we're going to subtract using saturating sub,

and that will be subtracting

the employee account start time.

Okay. Now we're just going to do

a little bit of error handling first.

So we wanna check to see that there is a total vesting time.

So we're gonna do, if the total vesting time returns zero,

then the vesting period wasn't invalid for the employee.

So let's just do that check first.

So if total vesting time is zero,

and this is also going to help with calculations

in the future because we don't want to divide

by zero anywhere.

So if total vesting time is zero,

it will return an error

and we will once again write a custom error code.

So we'll do ErrorCode, InvalidVestingPeriod, we'll call it.

And now we'll just add that to our custom error enum.

So we'll go down to the error codes enum,

and add InvalidVestingPeriod.

And we'll add the message. Okay.

And we'll go back to our instruction logic.

Okay, so now we wanna know the total vested amount

of tokens, which is the total amount of tokens

that the employee can claim.

So we'll do let vested amount equal.

Now, if the current time is after the end time,

then all of the tokens have unlocked.

So let's do a check for that.

So if now is greater than or equal to the end time,

then we're going to just return the total amount.

And then else, we're going to do a match statement,

and that's because we wanna make sure

that all of the calculations are safe,

and we're handling possible errors.

So we're gonna type it out like this, we'll do match, and then we'll take the total amount and multiply it

by the times and start.

So employee.total amount, oh, sorry,

employee account.total amount.

And we're going to use checked multiply.

So checked multiply is a multiplication

and the checked prefix allows you

to handle possible overflow errors.

So checked multiply,

and we're multiplying by the times and start.

Because the times and start is an I64 type,

because it's in Unix time,

and the total amount is a U64 type,

we're going to cast the I64 as a U64.

And you can do that by just typing as U64

because whenever you perform any calculation

against variables in Rust,

they have to be of the same type.

Now we'll do some, if the product of this multiplication,

we're just going to divide the product

by the total vesting time.

So what we're doing is we're taking the total amount, multiplying it by the times and start and then dividing that

by the total vesting time.

Okay. And then if there's a none value,

so the reason we're able to do this

is because checks multiply if there is an overflow error,

instead of panicking, it returns a non-value.

So here we're just doing a check in our match statement,

if it returns a value, we're going to be able to divide.

If it returns a none,

we're going to have to handle that error.

We'll just return an error

and we'll make another custom error.

And we'll name this one CalculationOverflow.

And then we'll go back down to our error enum

and add in the calculation overflow error

and add a message. Okay.

Okay. One second,

my VS code is being a little slow to update,

so we'll just let that update.

Okay, so just to recap this calculation,

if we didn't perform a match statement here to check the product of your multiplication,

then this checked mul is just going to return a none value,

and then we wouldn't be able to do the division

by the total vesting time.

So by implementing the match statement,

you're able to check to see if there's a product returned,

and if there is, then you do the division.

If there isn't, then you are handling the error

of a none value being returned.

Okay, so that looks good to me.

Now we wanna find what the claimable amount is for the user.

So we found the vested amount, so this is how many tokens

at the current point in time are unlocked.

But if a user came in earlier and already claimed some,

we wanna be able to handle that.

So what we're gonna do is find the claimable amount

by taking the vested amount and subtracting

the total withdrawn value that's stored

in the employee account state.

So we can do let claimable amount,

so we'll do the vested amount,

and then once again, use saturating sub

and we'll do the employee account total withdrawn.

Okay. And now a little more error handling.

If the claimable amount is zero,

we wanna be able to let the user know that they have nothing

to claim at the current point in time.

So we'll do let's, oh, sorry, we'll do,

if claimable amount equals zero,

then we're going to return an error of,

I will do the error code,

and we will make a custom error called NothingToClaim.

And then we'll update our error code enum down here

with our new custom error.

So VS code is being a little slow to update.

All right, there we go.

Now the error went away.

Now we did all the calculations needed,

so we now know what the claimable amount is for the user.

And now that we know what that is,

we can actually process the claim.

So by doing that, all we're going to be doing

is a transfer of SPL tokens from the treasury account into the employee's token account.

So we can do that by making a CPI call.

A CPI call, it's a Cross Program Invocation

and it allows you to make calls from another program.

In our case, the transfer is from the system program.

So we can do let transfer cpi accounts,

and we're going to define all the accounts needed

for this transfer to take place.

Here we're going to use TransferChecked,

and we wanna make sure this has the correct import,

which VS code is once again still being slow.

So instead of using the quick import,

we're gonna manually import.

Looks like there was a weird import generated. Okay.

So all we're gonna do is update our token interface

from the anchor-spl crate to include TransferChecked.

Okay. Now we can go down here

and you can see from the structure we need from mint

to an authority.

So we'll define all of those fields.

From, so the tokens are going to be coming

from the treasury token account.

So we'll load in the contact accounts treasury token account

to account info.

Now the mints is the next parameter,

and we'll just do the mint account.

And then we have to and we have the authority.

And in this case,

the authority is going to be the treasury token account.

And that is because when we earlier initialized

the treasury token account,

when we use the token constraint on the account attribute,

we specified that the authority was itself.

So here we can just set the authority to itself.

Okay. So those are all the accounts

that are needed for the transfer.

Now we can define the CPI program.

So let cpi program equal,

and that is going to be the token program.

So the transfer instruction that we're calling

from the Cross Program Invocation

is coming from the token program.

The next thing we're gonna have to do is define the CPI context.

But first, to be able to sign the CPI context,

we need to define what the signer is.

And in our case, we had a PDA

for the treasury token account.

So we need to now derive that PDA

to create our signer seeds.

So let's do signer seeds.

And the signer seeds have an interesting type.

So we're going to explicitly define that type here.

And all it is, is a few references.

So 1, 2, 3, and a U8 type. Okay.

So now we can go back to what our seeds were,

which is for the treasury token account when we created it.

Let's go to that context. Right here.

So I'm just gonna screenshot this, so we can reference it

and make sure we have the correct seeds.

So we first had our vesting treasury,

vesting treasury and then the next field

was the company name.

So we can load that in from our context.

So we can do context.accounts.

And the company name was saved in our vesting account state,

so we're going to use the vesting account.

And then within the vesting account was the company name.

And then lastly, we just need the bump defined.

So we're going to reference the context.accounts.

This is for the treasury accounts,

and we saved the treasury bump

into the vesting account state.

So we're going to call in the vesting account.

And then from that we're going to pull in the treasury bump

that we saved to the state.

Okay. So one thing we were missing was adding a reference

to the seeds. So we'll add that in.

And now we'll just double check.

We have our initial reference,

we have the second reference for the seed,

and then we have the last reference for the bump.

Okay. And actually the bump lives inside of this.

Okay. There we go.

So now we have three nested references,

and that is our signer seeds for the treasury token account.

Now that we have that, we can define our cpi context,

And this will be a CpiContext new,

and the first parameter is going to be the CPI program.

And then we need the accounts,

and now we need to specify that it's with signer

because it is a PDA.

So we need the signer of the PDA, so with signer,

and this will be the signer seeds that we just derived.

Now lastly, we need the decimals of the SPL token

that we're transferring and that we can just take

from the mint account that's passed through.

So we can do context.accounts.mint,

and then decimals. Okay.

Now we can actually process this CPI.

So we're going to use the token interface

and we'll do transfer checked

and we'll pass through the cpi context,

the claimable amounts and the decimals.

And now let's make sure the claimable amounts

is of a U64 type.

So we'll just do, we'll cast as a U64.

Okay. It looks like the signer seeds type may be off.

So we're just gonna update the spacing here,

so it's easier to read.

So we have our signing seeds and we have our reference.

Oh, okay. We're just missing a comma here. Here we go.

Okay. Now we need to import the token interface.

So the token interface

is coming from the spl-token interface.

We'll update the interface here.

So this should just have self. Okay.

Now we performed our CPI of transferring tokens

from the treasury token account

into the employee token account.

And all we have left is just to update the account state

for the employee to reflect that we've withdrawn tokens.

So we'll add employee account.total withdrawn

and increment that by the claimable amount.

And there we go.

So that is everything for our instructions.

Now, one thing to notice is in our parameters,

we have company name and it's returning a warning.

And that warning is because the company name is not being used within the instruction logic.

But we need to tell Rust that we know it's not being used,

but we're using it somewhere else.

So to do that, we're just gonna add an underscore here.

So just to recap, we've created vesting accounts

for the employer.

We've created employee accounts for the employee

to be able to claim their allocated tokens,

and we've created an instruction for employees

to be able to come to the UI and grab the tokens

that are claimable.

So now let's do an anchor built

to make sure that the program builds.

So we need to remember to be in our Anchor folder

to do any instructions that are related to Anchor.

Okay, so we can run anchor build.

Okay, so we can see that the code compiled,

everything's good to go.

There was a warning generated,

so this is just trying to help with formatting.

So we can go and make that update with the warning.

If we go to our match statement,

there were some parentheses here that just aren't needed,

so we can get rid of those.

Clean that up a bit. There we go.

Okay, so we can just one more time, run anchor build.

It'll be a lot faster now that it's already compiled once.

So now that we've built our new program

and everything compiled correctly, we can start testing.

But first, let's go over everything that happens

within this anchor folder.

So when you run anchor build, it updates the target folder.

So you can see here the target folder is now green,

meaning there are new files generated.

And we can go into our IDO,

and now there's an IDO created

for the vesting contract that we just wrote.

We have this counter IDL from the previous counter program

that was generated with the scaffold.

And we can just delete that because we don't need it anymore

and we will just keep the vesting IDL.

Now, there was also a types file generated.

We can once again delete the counter file.

And what this vesting types, it's a TypeScript file

that's compatible with your TypeScript

that uses the program IDL.

These two are going to be using for both our testing

and our front end application

because both of those are gonna be written in TypeScript.

Now all of that's done, so let's start with our test.

There is a counter test that's already generated.

We're just gonna delete that

because we no longer have the counter program

and we're not writing normal TypeScript tests.

We're gonna be using bankrun here.

So let's get everything set up for our bankrun tests.

So what this does is it creates a bank client a bank server,

and runs our tests against it.

Now we're going to have to specify the path of our program

to be able to deploy the program to the bank's client

and run the test correctly.

So let's make a fixtures folder inside our test folder.

And what that'll do

is we'll just import our vesting.solfile.

So if we go back to target and we go to deploy, you can see our .so is generated.

So we're just gonna copy that

and paste it into the fixtures holder.

And this just allows the bankrun test to be able

to easily access our .so file of the smart contract

that we just wrote.

And just to clean up a little,

you can see the counter keypair, anything counter related,

if you see it, just delete it.

Okay. Now inside our test folder we can create a new file

and that's going to be our bankrun test.

So we'll do bankrun.spec.ts. Okay.

Now there's a few packages

that we're gonna wanna install to be able to use bankrun.

So in our anchor folder, we're going to npm install.

And we want, let's see, what do we already have?

We'll go to our package.json.

And we have Anchor, we have Web3.js.

So now we also wanna add in the SPL token.

So we'll do @Solana.spltoken.

Okay, so you can see there's some dependency issues.

So what we're going to wanna do

is define a specific version.

So we're going to do @Solanaspl-token,

and we will do 0.4.8,

and we're going to update our Web3.js

to 1.94.0.

Okay. Great.

Now we're going to install anchor-bankrun.

Okay. And that'll be version 4.0.

And now we're going to install Solana bankrun as well.

Okay. And that'll be version 2.0.

We're gonna update the version here.

So this is just making sure we have compatible versions

for all our dependencies.

And then because we're going to be using the spl-token

when we're testing, we want to also use the spl-token

from bankrun.

So we'll npm I spl token bankrun.

Okay. And this one, we're also going to update the version

to 2.5 instead of 2.6.

So let's see, we have our coral anchor,

we have Solana spl-token version 4.8.

We have Web3.js version 1.94.

We have anchor-bankrun,

Solana bankrun, and spl-token bankrun.

Okay. So those are all the dependencies we need

for our bankrun tests

and let's just make sure everything's properly installed.

Okay, great. Now we can go back to our bankrun file

and start writing our tests.

So first thing, we're going to just set up our tests.

So we'll do describe,

and then we'll name the test vesting smart contract test.

Okay. Before we start actually running the test,

we first wanna start with our environment set up.

So in our environment,

we wanna make sure we have an spl-token mint

that we're going to be using with the proper authorities.

We also want to make sure we have the right programs

deployed to be able to test our best in contract properly.

So let's set up our environment first.

So we're going to do that by using beforeAll.

And what this is gonna do is run these specific functions before every single test is being ran.

So this is gonna be an async function. Okay.

Now within the bankrun test set up,

you have a wallet with a payer that is already funded

and ready to be able to work.

But in our smart contract,

we have two different scenarios that are being tested.

We have the employer perspective,

as well as the employee perspective.

So we'll keep the default native wallet

within the test to be for the employer,

but now we have to set something up for the employee.

So let's name this beneficiary,

and we're just gonna generate a new keypair.

So we'll do new anchor.web3KeyPair.

Okay. So we need to first define the type.

So outside of our beforeAll,

we're just going to let beneficiary be of a type, Pubkey.

Okay. And this is TypeScript.

So the type is actually public key, not Pubkey.

We're not in Rust anymore.

PublicKey. Okay.

Now let's fix our imports.

So we're going to make sure we import public key

from Web3.js.

We also want to import Anchor.

So we'll define our first import here as anchor.

And we're gonna fix Copilot suggestion.

And this should be from Coral X, Y, Z anchor.

(keyboard clattering)

Okay. And now we have a typo in key pair, so let's fix that.

There you go.

And this is actually a Keypair, not a public key.

So there you go.

Now one thing to note, the difference between a key pair

and a public key.

The key pair has both a private and public key attached,

so you're able to sign with it.

And because we need the beneficiary

to be able to assign instructions,

it needs to be of a type key pair

to be able to sign those instructions, okay.

And we're just updating our imports.

We'll just keep the public key import there

because we will be using that in the future.

So we've defined the beneficiary,

so next we're going to define our context for bankrun.

So we can do that by await startAnchor.

Now we'll define what the context is,

and the context is your program test context.

So program test context, and we'll import that.

So the quick import isn't showing up,

so we'll manually write that.

This is gonna be an import from bankrun,

so we'll do import ProgramTestContext from,

and this is gonna come from Solana bankrun.

Okay. So we'll be defining our context.

We're gonna do that with startAnchor.

And what startAnchor does is it starts a bank run

in an anchor workspace

with all of the workspace programs deployed,

and it spins up a bank server and a banks client,

deploys the programs and all the accounts that are added.

So let's import startAnchor,

and that's gonna come from Solana bankrun.

And let's update our program test context

to import correctly.

Okay, there we go.

Now we're going to update all of the parameters

that need to be passed through for startAnchor.

So startAnchor first has the path, which is the path

to the root of the anchor project.

And because we created this fixtures file

with the vesting.so pasted into it,

we're able to just make the path an empty string.

And now the next parameter

is going to be the extra programs.

So the program that we need to deploy

to bankrun is going to be vesting.

So we're going to set the name as vesting.

And one thing to note when you're naming these programs,

it needs to be whatever the name is of your .so file.

So you have to make sure that those two values match

or it's not going to pull the right program.

Okay. So we have the name and we also need the program id.

So program id, and this is going to be of a PublicKey type.

Now, when we created our lib.rs,

you have this declare id with a public key.

This is your program id.

When you run an anchor build, it creates an IDL.

And within the IDL, you have an address.

These two values should be the same.

You can see that they are, and this is the program id

of the smart contract that we just wrote.

So we can go back here and just pull that from the IDL.

So we wanna properly import the IDL.

So we will import IDL from,

and it's going to be our target IDL vesting.json.

That's the file path to get back to this file here.

Now in our program id, we can just call that address.

So we can do IDL.address.

Okay, and now we have a little typo.

This should be on the other side.

Okay, we passed through our perimeters

for the added programs, now we need to pass through

what our added accounts are.

Now, we talked about earlier how we need

to add a beneficiary account.

With bank run, you already have a signing keeper

that has both the public and private key

with funded lamports.

And we're gonna be using that for our employee.

But now we have another perspective

that needs to be able to sign instructions,

and that's our beneficiary.

So we generated the key pair for the beneficiary,

but now we need to load that key pair in

and make sure that it's funded with lamports.

So there are lamports to be able to pay the gas fees

when executing an instruction.

So we can define that here.

We will first specify the address,

and that will just be the public key of the beneficiary.

And now we'll add in the info for this account.

So this is just your typical info data structure.

It's going to have the lamports,

and we'll just fund it with a bunch of lampports.

And then it's gonna have the data,

and then it'll have the owner.

And in our case, it's just gonna be the system program.

So we can actually import the system program id from anchor.

And lastly, the executable, which we're gonna set to false.

And now we're getting an error here.

And that's because we have a typo in ID,

so it's just Camel case programID.

Okay, so that's everything we need for our context.

And just a recap, we have our startAnchor

that has the path to the root of the anchor program.

We also have the programs,

which is an array of the objects indicating

which extra programs need to be deployed

to the test environment.

And we have accounts, which is just an array of accounts

indicating the data that needs to be written

to a certain address in this testing environment.

So now we set up the context, now we need the provider.

So after context, we're going to define the provider,

and this will be a new bankrun provider.

And the parameter is just the context

that we defined in the previous step. Okay.

Now we need to make sure

that first provider's type is defined.

So provider is a BankrunProvider type.

And then we just need to import BankrunProvider.

And this is coming from anchor-bankrun.

So we'll do import Bankrun Provider

from anchor-bankrun.

Okay. Now that we have the provider,

we need to set the provider.

So we're going to do anchor.setProvider,

and that will be the provider that we just defined.

We have the provider, we set it,

now we need to define the program.

So we have the program and that's just a new program

and we wanna make sure that it is matching the vesting IDL.

So we'll add IDL as Vesting,

and we can import that vesting type

from our TypeScript file that was generated.

And the other parameter is the provider.

Okay, so first let's define the program type.

So program is Program Vesting.

We wanna import program from Anchor.

Now we wanna make sure, we already did an import of the IDL,

but we also want the TypeScript compatible version.

So we're going to import vesting

from the target types vesting.

And this is our TypeScript file.

There's two things that we're gonna be using a lot

throughout our tests, so we're just gonna simplify that

by creating a variable for that.

So one is the banksClient.

So the banksClient...

So one is the banksClient,

and the bank'sClient comes from the context.

So we can name banksClient is the context.banksClient,

and now we'll define the type above.

And that is a banksClient type.

Okay. And the next thing we need is the employer.

So we talked about how the provider comes with a wallet

that is already funded and has the full key pair

of both the public and private key.

So we can just name that employer,

so we don't mix up our two signing keys that we have.

So we have provider.wallet and then .pair.

Okay. And then we'll define what the employer is up here,

and that will be a key pair

because it is a signing key just like the beneficiary is.

Okay. So, so far we've set up our environment

with the needed accounts and programs,

but we also need to have an SPL token mint

that we're going to use as the token

that we're gonna be passing back and forth

between our treasury accounts and employee accounts.

So let's actually create a mint,

and we need to create that.

So we have the mint authority to be able to mint tokens

to the correct accounts.

So we will set mint,

and we're going to await and we will createMint.

Okay. And this is an import from SPL token bankrun.

So we'll add that import, import, createMint.

From, this will be spl-token bankrun.

There you go. Okay.

So now the arguments that are passed through

for createMint, we have the bank's client, we have the payer, we have the mint authority,

the freeze authority, decimals, key pair and programID.

So let's define all of those.

And first, before we do that,

let's actually define the mint type.

So we'll go back up here and do let's mint.

And the mint is gonna be a public key.

Okay. So now we have the banksClient.

Okay. And then up next is the payer.

And in our case, that will be the employer

that we defined earlier.

And then we want the mint authority.

So that is gonna be the public key of the employer.

For our testing purposes,

we don't need to freeze any tokens,

so we're just going to leave that as null

for the freeze authority.

And then for decimals, we'll just put two.

And I have a typo in banksClient, so let's fix that.

There we go.

I have a typo everywhere in banksClient,

so we'll fix all of them.

Okay, now there's an error,

there's a type error with the banksClient,

and you can see here that it is between the Solana bankrun

and the spl-token bankrun.

So this is actually within the dependencies themselves.

So what we're going to do is just do a TS ignore

because it does work between the two.

So we'll add in @ts ignore.

Okay. Now there's just a few more things to do

to finish setting up our test environment,

and then we can go to writing the actual instructions

to test the smart contract.

So first is setting up the provider

for the beneficiary key pair.

And we're gonna be using a different program

when we're actually executing the instruction

for the instruction that the beneficiary needs to sign.

So we'll just get that set up now.

So we'll do beneficiaryProvider.

So that is a new BankrunProvider,

and we wanna use the same context that we defined earlier.

So we can just do context. Okay.

And then we're going to do the wallet.

So beneficiaryProvider.wallet equals,

and this will be a new Nodewallet

of the beneficiary.

Okay. Now we need to define

what the beneficiary provider type is.

So we'll go back up to the top,

let beneficiaryProvider,

and that will be a bankrun provider.

And now we need to import NodeWallet,

and this is coming from anchor.

Okay. And then we can just define this second program

with the new beneficiary provider.

So program2 is equal to new Program of type vesting.

And then we'll need the IDL as vesting,

and it's selling the provider as the beneficiary provider.

Okay. And then we'll once again define

the type for the new program.

So let program2 equal the Program Vesting type.

And lastly, we just need to derive the PDAs.

So we have three PDAs throughout our smart contract, and we'll just derive all of those now.

So we'll do the vestingAccount key,

and now you're able to find the address from the PDA

by just using the find program address sync.

And what this does is it generates the public key

for a specific PDA.

So PublicKey. and we'll use findProgramAddressSync.

And then we need to define our PDA.

So for the vesting address,

we used just the company name to create the PDA.

So we can do buffer.from company Name.

And then we just need the program id.

So we'll do program.program id.

Now two things.

One is defining the type of the vesting account key.

So we'll go back up to the top,

let vestingAccountKey is a PublicKey.

And then the next thing is the company name.

So since the company name is a user input value from the UI,

we're just going to explicitly define the company name

for testing purposes to make things easy.

So we'll make a constant at the top.

Company name is just...

We'll make a company name for now.

Okay, cool. So that's our vesting account key.

Now we need a treasury token account key.

So treasuryTokenAccount.

And we're gonna do the same exact thing.

So PublicKey.findProgramAddressSync.

And this is where we define the seeds for a PDA.

So we'll do Buffer.from.

And for our treasury token account,

we did vesting treasury as the first part of the seeds.

So vesting treasury.

And then the second part was the company name.

So buffer from companyName.

And then lastly, we just need our program id.

Okay, we've forgot an equal sign, so we'll add that.

And we need to define the treasury token account type.

So let treasury token account be a PublicKey type. Okay.

And the last PDA we need to derive is the employee account.

So let's define that.

So same exact method.

We have our EmployeeAccount,

and that is a Publickey.findProgramAddressSync.

And then the seeds for our employee account

was a little bit longer than the rest.

So we have buffer from and it was employee vesting.

Okay. And then we had the public key of the beneficiary,

And then we had the vesting account key.

So we had a typo in public key.

So just for a reference,

the public key class has a capital K in TypeScript.

Okay. And then lastly, we just need to define

the program id.

Okay. Let's just fix some formatting a bit.

So this is a bit easier to read.

Okay. There we go.

And now we'll once again define the type.

So this will be let employeeAccount be a public key type.

Okay. So that's everything for the PDAs

that we need to derive.

And it looks like we have everything set up

for a testing environment and we can start actually testing.

But let's just recap everything that we wrote.

So beforeAll sets up the environment,

and this is gonna be ran before every single test

that's ran.

And here we're setting the beneficiary key pair.

We're running startAnchor,

which is setting your bankrun in the Anchor workspace

and it's creating the banksClient and the bank server.

It is specifying the path to your .so

of the smart contract that we're testing,

it's specifying the programs that we need to deploy,

which in our case it's just the vesting contracts.

And then it's also specifying the accounts

that need to be funded and have specific data written to it.

And for our case, it's just funding the beneficiary account.

We're setting our provider running anchor setProvider,

naming our program, setting the banksClient.

We're creating a mint address

with the employer being the authority.

We are updating the provider for the beneficiary.

And then we're getting all of our program addresses

for each PDA that we need to derive with the seeds that we're specifying.

Okay, great. So now we can start writing our test.

So we'll go outside of the beforeAll

and write our first test.

So this should...

We're going to create a vesting account. Okay.

And this is going to be an async function as well.

And now we can write our first transaction.

So we'll do const, tx and await.

And now we can just call the namespace from the program.

So we'll do program.methods,

and now we can call the create vesting account instruction.

So create vesting account,

and all we're passing through as a parameter here

is the company name.

So we'll write companyName,

which is a constant that we defined earlier.

And now we need to define all of the accounts

that need to be passed through to run this instruction.

So when we're creating a vesting account,

we need the signer, we need the mint,

and we need the token program.

So we're going to set accounts,

and signer is going to be the employer,

but it's gonna be the public key.

And then the mint is just the mint that we already defined.

And then the token program is,

and this is just the token program id, there we go.

So with the new version of Anchor,

any account that is a PDA that needs to be derived,

it automatically does that for you.

So you're just passing through the additional accounts here.

Okay, so now we need to send the transaction.

So we'll do .rpc and then the commitment is confirmed.

Now we're just going to console log our transaction.

So we'll do a console.log and then Create Vesting Accounts

and it will be our transaction.

So another thing we can log actually

is the vesting account data just to make sure

that the vesting account was created correctly.

So let's do const.

We'll do vestingAccountData,

await program.account.vesting account,

and we can fetch our vestingAccountKey.

Okay. And then we'll just console log that.

Okay. Now we can run our first test,

make sure it works.

So in our Anchor folder,

all you need to do is run Anchor tests

and it will run all of the bankrun tests.

So you can see it's recompiling the Anchor program,

and now it's running the test suite.

And you can see our test passed.

So let's go back to our logs within anchor-bankrun.

So here's our test.

It's running the bankrun spec, TypeScript file,

it's running Solana program test.

Here are your runtime messages.

And then here's our console log.

So the vesting account data,

we have the owner is the public key.

We have the mint address,

we have the treasury token account, the company name,

we have the bump for the treasury

and the bump for the vesting account.

And then we also have the transaction

of our create vesting account.

And you can see in our test suite, we have one test passed

and the test was named showed create a vesting account.

Okay, so now we can go onto the next test.

So in our first test, we created the vesting account.

Now the next instruction that we wanna test

is creating an employee account.

So let's write that test out.

We'll do, it should create a...

Actually, before we create the vesting account,

we need to fund the token account.

So let's think back of what we just did.

So we created a vesting account.

And when the vesting account was created,

not only was the account that's holding the state

for the vesting account created,

but a treasury token account was created as well

because when we wrote our context data structure

for creating the vesting account,

we had two initializations.

We had the initialization

for the vesting accounts and the initialization

for the token account for the treasury.

So this token account needs to be funded to be able

to pass tokens from the token account over

to the employee's token accounts later on in our tests.

So since the SPL token mint was created

with the employer as the mint authority,

we're gonna be able to mint tokens to the token account

that was just created to fund it.

So let's add that as the next step as our test.

Should fund the treasury token account.

So let's define the amount that we wanna fund.

So this is gonna be in lamports,

so we'll do 10,000 times 10, oops,

times 10 to the ninth. Okay.

Now we can just execute the mint transaction.

So we'll do mint tx await,

and we're going to run mintTo. Okay.

So we wanna make sure that we import this

and we're gonna update the import

from the spl-token bank run.

Okay. Now this has a few arguments.

It's very similar to createMint.

So first we wanna pass through the banksClient.

This is gonna generate the same type error from createMint.

So we can just copy over our little comment here.

Okay. And now we have employer

as the next argument, mint,

and then the destination is going

to be the treasuryTokenAccount.

And we can just pull in the key that we derived earlier,

and then the employer is going to be the signer.

And then the amount is going to be the amount

that we just specified above.

Great. So now we can just console log that transaction

and let's run Anchor tests and make sure that went through.

Okay. So you can see our whole test suite passed,

which included two tests and those both passed.

So now our treasuryTokenAccount is funded.

So we can move on to the next step.

And this is going to be creating

an employee vesting account.

So now we can just define our transaction.

And this is gonna be the second instruction of our program.

So we'll just name it tx2,

and we're going to await, we'll call program.methods.

and we'll do create employee account.

Now this has several parameters to be passed through.

So let's see.

It has the start time, the end time,

the total amount, and the cliff time.

And because these are both I64 and U64,

those convert to big numbers in TypeScript.

So we're going to create a new big number

and we'll do just for simplicity,

we'll set the start time to zero

and the end time to a 100.

We'll set the cliff time also to zero,

and the total amount to a 100, so total amount is next.

And then the cliff time to zero.

This just helps us with testing

because it sets the, sorry,

so we don't have to wait for the cliff to pass.

Okay. So now we'll just import BN from Anchor.

Okay. So next is passing through the accounts.

So accounts and we'll need the beneficiary

and the vesting account.

Okay. And lastly, we'll send the transaction,

so rpc commitment confirmed,

and we're also going to skip pre-flight to true.

So. There we go.

So we'll console log this transaction,

make sure it went through.

Okay. And then we'll just confirm the Employee Account.

So we'll just console log the Employee Account address,

and this is gonna have to be tobase58.

Okay. Now we'll run Anchor test one more time.

Okay, so you can see the test we passed,

all three tests have passed and it names each test.

We can just go back to our console log

to just check everything.

So we have our first instruction,

and it created the vesting account.

And there's our transaction signature.

We have creating an employee account,

and here's the transaction signature.

And then this console logs the employee account public key.

And we also have the minting tokens

to the treasury token account.

Now the last thing we need to check is claiming tokens.

So we'll make one more test.

So should claim tokens.

(keyboard clattering)

So now, because we need time to pass here

to be able to claim the tokens, we're going to use a clock.

So we're going to const currentClock

is equal to await and from the banksClient,

we can just get the clock.

So banksClient.getClock.

Okay. And now we can set the clock in our context.

So conts.setClock,

and this will be a new Clock with,

and we're going to use the currentClock slot.

And then currentClock epochStartTime.

And I spelled currentClock wrong.

Let's fix up.

Okay, and then let's make sure we import clock.

So this is an import from Solana bankrun.

Okay. And now we can see all of the arguments needed.

So clock start time, we need epoch, next.

So we'll do currentClock.epoch.

And then next is the leader schedule.

So currentClock.leaderScheduleEpoch.

And then last is the Unix timestamp.

So we'll just set this to 1000

and okay, now our clock set.

Now this is gonna take a little bit longer

in our bankrun test,

so we're just going to set a time out.

So we'll do await new Promise,

resolve setTimeout.

Okay. Now lastly, we're going to run our third instruction

of the smart contract.

And here we're actually gonna use the second program

because the signer for this instruction

is going to be the beneficiary and not the employer.

So methods and we'll call claimTokens.

Okay. So what we're passing through here is just the company name.

Now we need to add the accounts.

And for this, it'll just be the token program.

And then we need the RPC commitment confirmed. Okay.

And now we'll just console log our transaction.

And this is our last test to run.

So we'll run Anchor test

and see if the employee was able to claim their tokens.

Okay. So you can see all four tests have passed,

the full test suite has passed.

And we can just look back at our console logs

from the beginning of our bankrun test.

So we have the vesting account.

When we're creating the vesting account,

your transaction was logged,

we have the minting to the treasury token accounts,

we are creating an employee accounts.

Here's the employee account public key,

and then here is our claim tokens transaction.

Okay, so now that our tests have worked

and we have our program built,

we're gonna be able to deploy our program.

So in this whole Bootcamp,

we're going to be using our local testing environment.

So let's get a local test validator running,

and then we can deploy to it.

So we're gonna go back to our terminal

and let's just do solana-test-validator.

So running the Solana-test-validator

just initializes a new test ledger.

And then is running a local validator on your computer.

Now that this is loading, you can see that it is loading

to my local host and we're able to now use it.

So once you build a program, you're able to deploy.

So we can just run Anchor deploy,

and we're going to specify the cluster

that you wanna deploy to.

So we'll do provider.cluster,

and this will be our local host.

Sorry, with Anchor, it's localnet,

with Solana, it's localhosts.

Okay, there we go.

So you can see it's deploying vesting.

And you can see this is our program id.

Now one thing to check

is you wanna make sure all of the program ids are synced.

So we're gonna run anchor keys sync

and make sure we're using the correct program id

throughout the entire workspace.

Okay. So you can see all program id declarations are synced.

There was an incorrect public key in our anchor.toml.

So this was updated by running Anchor key sync.

So just because there was an incorrect program id,

we're just going to go back and rerun anchor deploy,

make sure everything's correct throughout the workspace.

Okay, great. We're going to start working on our front end.

So we can take this program ID and connect it

to our front end.

So in the Create Solana dApp Scaffold,

there is one part that doesn't automatically update

the program id and you have to manually update it.

So we're gonna go there and make sure

that it's using the correct program id.

So that's here in our exports.

And we actually need to fully update our exports

because it's still working for our counter.

So we're going to update our counter exports

for our vesting program.

So let's rename this to vesting exports.

And what this is doing is it's exporting information needed

to be able to pull into our web folder

when we're creating our web app.

Okay. Now everything that's counter we wanna update.

So this is actually our vesting IDL,

and this is coming from IDL.vesting.json.

Okay. And we're going to import the vesting type

from our target types vesting file.

Okay. So now we can export our vesting investing IDL.

Now we'll update the program id for vesting.

So vesting program id,

and this will be the VestingIDL.

(keyboard clattering)

And now this public key should be the same

as the program id here.

So we're just going to paste this in.

Okay. And this will be the Vesting Program ID.

Now we're going to go back into our files

and just check for counter.

Okay, great.

So you can see everything left with counter

is in our React app, so we can worry about that

as we're building out the front end.

Now we can close our anchor file and open our web.

Okay, so if you look on our web folder,

we have the app folder, we have components,

and we have public, we're gonna start out in components.

So you can see the components are separated

between account cluster, counter, dashboard, Solana, ui.

What we're updating is the counter 'cause the counter

is the area that's connected to the anchor program.

So let's rename this to vesting.

Let's also rename this to vesting.

Okay. Just for consistency.

Now you can see the vesting component

is split into three sections.

We have data access, we have feature, and we have ui.

Let's first work with data access,

'cause that's where we're going to be accessing

all of the information needed to display on the ui.

So we're going to import our getVestingProgram this time

and getVestingProgramID.

Okay. We'll just update everything

from counter to vesting.

Okay. So when you're using the vesting program,

it's going to be making a query.

And what we're going to wanna set up is a query

for the vesting accounts.

So we're updating that.

Now it's getting the program accounts

based on the program id.

Now here's the initialization.

We wanna update this

to initialize our new vesting accounts.

So let's remove the initialization function.

And this is just coming from the counter program,

you had to initialize the counter program.

So let's make this create vesting.

Actually, let's keep this just to follow the same format

and we're just gonna rename it.

So we'll do createVesting Account.

Now this is program.methods,

and it's gonna be your createVestingAccount.

This is going to pass through a company name

and a mint address.

So we have our company name.

Now this is a user input

and we're pulling this from the front end.

And because we're doing that,

we're going to have to tell this mutation

that we're doing that.

So we're gonna update the mutation to have,

We have a string, we have an error,

and we also need to specify our arguments.

So CreateVestingArgs, and let's write an interface to define

what that is.

So we'll do interface, CreateVestingArgs,

and we will need the company name.

And we need this because this is a user input parameter.

And we're using this as an argument

for the create vesting instruction.

And this is its string type.

Now another thing that we're going to have pass through

is the public key of the mint.

And that's because we don't derive

the mint account anywhere,

so we need to pass through what the mint account is.

And that is an account

that's being used within the context data structure

when you are actually creating the vesting account.

But since it's being an input on the front end,

we're going to make the type of string

and later when you're using the value,

we're going to have to generate a public key from it.

So we're passing through the arguments that we need.

Our mutation key is going to be our vesting account,

creates, we're using the cluster.

Okay. So here is where we wanna pass through

the company name and the mint.

This is because these are the two arguments

that we're grabbing from the front end

and then passing through.

So we just need to specify that.

Okay. Now we can run program methods

and we're using CreateVestingAccount.

And the only parameter

that's being passed through there is the company name.

Okay. And then the accounts that we need

is going to be our mint account and the token program.

So we'll go back to accounts and we'll do the mint,

and that is a new public key of the mint.

And we're creating a new public key type

because the mint is being passed through as a string type.

And now we also need the token program,

Okay. And then we'll just import the TOKEN PROGRAM ID.

And then we don't need a signer

because we're going to actually be using the wallet

that is connected to the front end of the application.

So we'll just use the RPC and then on success,

it'll sign and on error,

it will give fail to create vesting account.

Okay, great. So now we'll just update their return.

And that is going to be create vesting account.

Okay. So that's all the data we need for using the vesting program.

Now we wanna be able to use a vesting program account.

So what this is gonna be is once an employer has come

to the front end and they have created a vesting program,

now they can go in and update that vesting program

by adding in employee accounts.

So we'll go to useVestingProgramAccount,

and this is passing through the account.

Okay. I will do useVestingProgram,

and then the query is going to be for the vesting accounts.

Okay. So now you can see we have close,

decrements, increment, set.

This is everything related to the counter,

so we can just delete that.

Okay. And we will delete the corresponding returns.

Okay. So now we're going to wanna add

in creating an EmployeeVestingAccount.

So this is gonna be very similar to our instruction above.

So what I'm gonna do is just copy this over

to make things easier.

Okay. Now instead of CreateVestingArgs,

we're going to create an interface for CreateEmployeeArgs, because these are gonna be different.

When you're creating an employee account,

you're specifying things like the start time

and cliff time of your vested tokens.

Okay. So interface CreateEmployeeArgs. Okay.

Now we'll define what that is.

We need the start time, which is a number.

We need the end time, which is also a number,

the total amount, which is a number as well.

And then the cliff time, which is also a number.

So those are our four arguments needed

when you're creating an employee account.

Okay. So we can go back to our CreateEmployeeVesting,

we'll update this to employee account

and then we're passing through

the arguments we just defined.

So start time, end time, cliff time,

and total amount.

Okay. And then we'll update this

to create employee accounts.

And that is gonna take in the start time, end time, total amount and cliff time. Okay.

And now the accounts, let's go back to our bankrun test

and just check to see what accounts we need.

So accounts, tests, bankrun.

So one good thing about testing in TypeScript

when you're creating the front end, it's very similar,

so it makes things a lot easier

'cause you can always reference your tests

and you know that these work.

For creating an employee account,

you need the beneficiary, and the vesting account.

So we'll go back to the data access,

and we're gonna update our employee args

to include the beneficiary, which is going to be a string

because it's going to be a user input.

Okay. And then we'll go back and do the same thing

to convert it to a public key.

So this will be beneficiary,

new public key of the beneficiary.

And we will update that here. Okay.

And now we don't need the token program,

but we need the vesting account.

And that is just the account.

So now we can return the CreateEmployeeVesting.

Okay, great. So for now,

this is what we need for our data access.

Let's update our UI and then the feature.

So let's rename this to vesting UI.

We need to update all of our imports.

So let's just go through everything here.

And we also need to update the data access file name.

So we will change this to vesting,

and while we're here,

we can update some of the unused imports.

We no longer using a key pair because we're connecting

to the wallet rather than generating a key pair.

And we're no longer using program.

Okay. Okay. So we'll update the create

to VestingCreate,

and then we're using the CreateVestingAccounts

from useVestingProgram.

Now this is what the UI is gonna look like,

as of right now it was just a button that you clicked for the counter.

However, we need to accept some user input parameters,

so we're gonna have to change that a little bit.

So first we're going to need the company and the mint.

So let's actually update this to have the company

and then set company.

And we're going to useState.

Okay. And we're just gonna import this from react.

And then we also need to do the same for mint.

Okay. And now because we're using the wallet

that's connected, we wanna be able to get the public key

from the wallet.

So const public key.

And this is going to be useWallet.

And this should be an import from the Solana wallet adapter.

And you can see that updated the import.

Okay, so now we have that set up.

Now we need to be able to get these two values from the UI.

So let's create a text field for both of those.

So in the return, we're actually going to create a div

and we'll put this button inside the div.

Okay. So now we'll have a text field.

So we'll do input and let's just take what Copilot has,

and we can update it as we need.

So we have the type as a text, good,

the placeholder is company name, good, the value is company.

And then on change, we want to set company.

Okay, great. Now let's also add in the class.

So we'll do class name,

and we're just going to define this as an input.

And we want to have a border,

we want the width to be full.

And let's do max width for extra small.

Okay. There we go.

So that's our first input.

Now we need another one for the mint.

Okay, we're just gonna accept

what Copilot's suggesting and then review it.

So we have the input type text, we have the placeholders,

the mint address, the value is the mint,

it's updating the mint on change,

and then it's taking the same exact class name from above.

Now let's update the button.

Well, we can change the name of the button,

and that'll be create New Vesting Account.

Okay. Now we're gonna need to do a little bit different

than how the counter is,

and that's because we have some values that are being input.

So we wanna make sure that those are added from the user.

So we're gonna actually check the form.

So let's do a few steps

before we actually update the button.

So first is we wanna make sure that the form is valid.

So let's do const, we'll do an isFormValid check,

and we just wanna make sure the company.length

is greater than zero and the mint length

is greater than zero.

Okay. Now we're just gonna do a handle submit.

So let's do a const handle submit.

And we're gonna do if, so we wanna make sure, one,

the wallet's connected, and two, the forms are valid.

So if the public key exists, and if the form is valid,

then it's going to run createVestingAccount.mutateAsync

with the companyName, which is the company and the mint.

Okay. And now we just wanna do one more error handling.

And that's just if the public key doesn't exist,

we wanna return an error to tell the user

to connect their wallet.

So we'll return, connect your wallet.

Okay. So now that those checks are in place,

we can just update what happens on click for the button.

So on click, all we're gonna do is just handle submit,

and then it should be disabled

if the createVestingAccount is pending.

And it should also be disabled if the form is not valid.

So a user cannot click the button

unless they've fully filled out both of the forms needed.

Okay. And then we will just update the button

to have createVestingAccount is pending.

And there we go.

So now we've updated the vesting create page.

Now we need to update the list.

So what this does is it returns a list

of all of the vesting accounts that are connected

to the connected wallets public key.

So if I created multiple vesting accounts because I'm separating my company

into separate mini companies

and I want a vesting account program for each of those,

it'll show all of them.

We're going to update this to have useVestingProgram.

And now what this is doing is it's checking the data

of the program account.

If the account is not found, it's going to return an error

and say you need to make sure that the program account

is deployed to the correct cluster.

Meaning if you're connected to LocalNet,

the program should be deployed to localnet.

If you're connected to Devnet,

the program should be deployed to Devnet.

So now you have the return and it has your cards.

So this is going to be all of the cards associated

with the vesting account.

So we're just gonna rename this to vestingCard

and then we will find where the counter card

was originally written.

Okay. And we're gonna update this for vesting.

So this should use vesting program.

Sorry, this should be used vesting program accounts,

and it takes in the account.

And now we'll update this to have createEmployeeVesting.

Okay, so this we're gonna make updates very similar

to the previous section because we're going to be taking

values from the front end

and using that in processing our instructions.

So let's set up all of the information that we need.

We'll need the start time.

And now instead of using state as a string,

this is gonna be a number.

So we'll update that to a zero

and we'll do the same for end time,

the same for total amount and the same for cliff time.

Okay. Now we're also passing through the company name,

but because we're connected to the account,

instead of needing to pass that through on the front end,

we can just query the account because the company name

is saved to the account state.

So we're going to do const company name as equal to,

and we'll do useMemo.

So we'll do an account query of the data in the account,

we'll unwrap that and find the company name.

Okay. And then we'll just need, there we go.

Okay, so we can remove this

because that's coming from the counter

and that was just querying the count from the counter state.

So we're doing basically the same thing,

but querying the company name instead.

So now we'll return, we'll update the UI here.

So we have, this is separating everything into cards.

And we have the header, this is just querying the account.

So from the account, instead of the count,

we want the company name to be the header.

Okay. Now, before the buttons,

we're going to need a text input.

So we're gonna do basically the same thing we did above.

So for simplicity, let's just copy that over

instead of having to write it all out.

Okay. So we'll make that update under the header

and then under this div,

so this div is separating all of our buttons.

So we have an input, and this will be text.

We're going to set this as start time.

The value will be start time,

and now we want the start time placeholder to show.

So because of that, we're just going to do this.

Okay. And now we have the value on change.

It's going to set the startTime.

And this is going to be...

Because this is a number,

we're going to have to parse the integer.

So we'll do parseInt, there we go.

And we'll keep the class name the same.

So now we're gonna do the same thing

for the rest of the inputs.

So we'll just copy this

and update this to endTime.

And this will be set endTime.

Okay. Then we'll do the same for cliff and total amount.

So total amount comes first actually,

and we will set total amount and then we'll do cliff time.

Okay. So let's just make sure we did all of that.

So Start Time, setStartTime, End Time,

total amounts and cliff time.

Okay. And let's just write allocation here,

so it's easier on the front end,

so you know what that value is.

It's the total allocation an employee gets.

Okay. So now we can update our button.

So the button, it'll have the same class.

On click, we want to createEmployeeVesting.

And that is going to update these arguments to startTime,

endTime, totalAmount and cliffTime.

Okay. And then we'll update this to createEmployeeVesting.

Okay. And now we're getting an error.

So let's figure out why.

So you can see beneficiary is missing in the type.

So we actually are forgetting one field.

So let's update this to also have the beneficiary.

So here we're going to need const beneficiary,

setBeneficiary and beneficiary

is a type string, so we're going to use the string.

And then we're gonna need to have an input field for that.

And this will be a beneficiary,

and this will be the wallet address. Okay?

And we'll update this to just beneficiary.

And this is not an integer,

so we can get rid of the parseInt.

Okay. And then this should be setBeneficiary. Okay.

And now we can pass that through as well.

Okay. And we'll update the name of the button

to create employeeVestingAccount.

And this we can get rid of

'cause this is coming from the counter program,

and we will just clean everything up, okay.

And we'll clean up our imports.

Anything unused we'll get rid of.

All right. So lastly, we need to update our feature.

So the counter feature is going to use the vestingProgram,

it's going to use the vestingCreate.

Let's update our counter list to be named VestingList.

Okay. Then we will update everything for vesting.

So this should be a vesting feature. Okay.

This should be a VestingCreate,

VestingList. Okay.

We will update the title to Vesting Smart Contracts.

Actually we'll just name it Token Vesting. Okay.

And then create a new vesting account below.

And we'll just remove the rest of these comments.

Okay. So let's see.

This is our feature.

We're setting up the VestingFeature.

We're connecting the public key from the wallet

that is connected to the front end.

We have the program id from our vesting program.

If there's a public key connected,

it'll overturn to create a token vesting account.

Okay. So this looks like everything we need,

we're just gonna rename this vesting.

And now we should have gotten rid

of almost everything related to the counter by now.

There's just one more thing we're gonna need to update.

So vesting, let's just command shift F for counter.

Okay, these are all just comments.

So here we go.

In the layout, we'll update this to Vesting Program.

And this should be using the vesting path.

On the page, we want to use the VestingFeature, and we're going to import the VestingFeature.

Okay. These are all comments. Okay.

And now let's just update our queries to vesting.

Okay, so now we're good to go.

So now to recap everything that we've done so far,

we've created our smart contract using Anchor

in our Anchor workspace.

We've built the project and deployed it also using Anchor.

We deployed onto our localhost,

so we're running a Solana test validator locally

on our computer.

And then we've updated the front end to be able to connect

to our wallet and execute smart contract instructions.

Now that all of that's done,

we're going to run on our local host the UI

and be able to test it out.

One thing to know before we start running,

you're going to want to make sure that your package.json,

the log file is updated

because it's probably still going to have dependencies

linking to all of your account or files.

So I would just delete the package.lock

and rerun npm install to make sure that everything is set

and good to go.

And then we can just go into our web folder.

So we'll CD into web, and we can run npm run dev

and get this running on our local host.

Okay. So we're gonna open up our browser,

Navigate to our local host.

Okay, so this is what our UI looks like

using the Create Solana dApp.

You have the cluster that you're connected to.

You can see here we're connected to local.

You have your wallet connection here,

and this is connected to my Phantom wallet

that I have linked on this webpage.

Now you can see that it automatically prompts you

that I'm on local, but I don't have an account found,

so I'm gonna request an airdrop.

So this account that my wallet is connected to

is now funded with SOL, and the account exists.

So this is our placeholder.

You have this basic front end that has just GM, this is your new Solana dApp and some links.

Now what we updated was the vesting program tab.

And just to make sure you're familiar

with the Create Solana dApp Scaffold,

we'll just go through all the other tabs first.

So you have this account tab,

and this shows the account that I have connected.

Here's my wallet address.

It is the same as the wallet that's connected.

You can see that I have one SOL

that I just airdropped to myself.

You can see the transaction history,

which was the SOL that I just airdropped to myself.

You can see that I have no token accounts currently,

and you can see that we have the option to airdrops

and receive funds from my wallet.

Now we can move on to the clusters tab,

and these are all the clusters that you can connect

to from your front end.

So you have Devnet, you have localnet,

and you have testnet.

Currently we're connected to local,

and we're running our local on our Solana test validator

in our terminal.

You can see our status of our current validator.

Here's the process slot, confirmed slot, finalized slot

and full snapshot slot.

So you can see that everything's running correctly

on our local host.

All right, so now we can go to our vesting program.

You have the name Token Vesting.

It says, create a new vesting account.

This is the programId of the account that we just deployed.

Now let's test this out,

but we're going to need an correct mint address

for this to work.

So we're gonna go back to our terminal, open a new tab,

and we're going to create a mint on our local host.

So let's do spl-token create-token.

I'm gonna specify the URL to be your local host.

Okay, so you can see we're creating a token

under the token Keg program.

So it's an spl-token.

Here's the address of the token we just created,

and there's the signature,

so you know it was a successful transaction.

So we're gonna copy that address, go back to our UI

and paste the mint address in the mint address field.

Okay. Now we'll do the company name.

So we'll just name this bootcamp.

Okay. So we'll create a vesting account.

Unlock, confirm the transaction. Oop.

Great, so now you can see the new vesting account

that was created for the Bootcamp.

And this allows the employer to now come in

because this Bootcamp card from the new vesting account

that was created is connected to the public key

that I'm connected to on the UI.

So all of the vested accounts

that I create are going to show up here.

And then me as the employer is able to come in

and create employee vesting accounts for employees

for each account that I create.

So if I wanted to make another one,

just like a subset of vested tokens,

and I set this as my 2024 Bootcamp employees,

we're going to just create this,

create new vesting account. Confirm.

Now you can see this is my other set

of vested allocated tokens. Sorry.

You can see this is my other set of vested tokens.

So now I'm managing two different accounts

that have separate tokens.

I could create a separate mint address if I wanted to,

so we can actually just test that out.

I'll go back to the UI.

I'm gonna run spl-token create-token.

Now I have a whole new token.

So let's say I wanted to have some employees vested

with one token and some with another,

I can do that.

And we can do just a test here,

create a new account, confirm,

and now you can see here's my next account.

So this is related to one token,

and this has a mint address of another.

So now I can keep track of all of the vesting accounts

that I create.

And within those, I'm now able

to create employee investing accounts

that have a start time, total allocation,

end time, cliff time, and the wallet

that I want the beneficiary of this account

to be able to claim tokens.

So there's my UI,

and you can see everything works with the program

that we created.

And overall, we created a smart contract.

We tested it using bankrun, we updated the front end

with our Create Solana dApp Scaffold,

and we tested it on the front end.

We also deployed our smart contract to our local host

while we're running our Solana test validator locally.

So there's our token Vesting dApp.

- Now that we've been building with tokens,

we're gonna add something new in our next project.

The next project will be a token lottery system where the user will be able to enter into a lottery

with a smart contract acting as the ticket issuer.

The hard thing about lottery systems in blockchain

is that they require some amount of randomness to be fair.

A distributed system coming to a decision

on a fair random number is impossible.

So we're bringing in a new concept

to help us out.

In blockchain, oracles can be used

to retrieve random numbers for use cases on chained.

Oracles connect the distributed system to the data

of the outside world.

Financial data, football game outcomes for betting markets

or even random numbers can be retrieved from these Oracles.

The way these Oracles work

is that they have a distributed set of data providers

from the outside world pushing data on chain.

The on chain data is then normalized

within your smart contract, throwing out data oddities

and taking a fair number based on the data provided.

Random numbers are still difficult, even with Oracles,

but they can be retrieved using a commit reveal scheme.

First, someone will commit a random number

that is hashed with a random seed value.

This commit value can be stored on chain for use later.

Once ready to reveal, the original committer will then push

both the random number and the seed value,

which can be verified against the original hash.

We're gonna do exactly this commit reveal scheme

in our project.

Let's get started.

All right, so for this project which we have called

the Token lottery, what we're going to do

is we're going to have a lottery system

that people can buy tickets from.

They can use those tickets to then claim prizes

and then we're gonna use something called

off-chain randomness.

So it's not actually on chain,

it's using something called Oracles to decide the winner

of the specific lottery.

So let's get started.

Before we do any code, what I'd like to do is I'm going to draw up all the different things

that we need to build.

So we're gonna design our program.

So this is the token lottery, right?

The token lottery is gonna have a few functionalities.

So we have issue ticket.

We are going to be able to, let's see,

we're gonna have to be able to do a commit reveal

for randomness and finally claim prize.

And so thinking about this,

what kind of instructions do we need?

So I'll start building out and designing a program.

So the first instruction we obviously need,

we're gonna have to create the configuration

for the lottery.

So this is gonna be like initialize config.

So that will build everything

that we need to for that lottery.

The next thing that we might need

is we might need a initialized lottery,

so that we can create all the different pieces

for creating our collection, being able

to issue tickets later, et cetera

'cause we wanna make sure that we initialize

the lottery collection first before we can allow people

to buy tickets.

The next one, this one's gonna be something

that's going to be done by the user of this,

but we're gonna do a buy ticket function,

so that people can buy those tickets

and then they can later use those to claim.

So we're gonna get into this a little bit later,

but we're gonna have two separate instructions.

One's gonna be a commit function,

so commit randomness function.

And the second one will be a reveal randomness,

or choose, we'll call this choose winner.

The reason being

is whenever you do like a commit reveal type of scenario

or scheme, you have to make sure that you do those

in two different instructions,

specifically also two different transactions.

You cannot do the commit and reveal both in the same transaction because there's some gaming

that you can do there.

And the final instruction that we'll need is claim prize.

So claim prize.

So you can see there we have six different instructions.

You can already kinda think

this is gonna be a pretty thorough and in-depth smart

contract that we're gonna be building.

Let's figure out what that configures.

So we're gonna call this the token lottery.

This is gonna be everything about the token lottery.

We're gonna need things like we're gonna figure out

who the winner is.

This will be a ticket number basically.

We're gonna have to figure out if the winner

is already claimed because what we don't want to happen

is someone to be able to claim the prize immediately

on Token lottery without anybody doing anything.

We wanna make sure that it's bound by the time

that the token lottery is taking and time,

meaning we have a start time and an end time

for the token lottery.

Next thing we have is we're gonna have

an actual winning amount.

So we have to have a,

we'll call it a lottery pot amount.

This is the amount of money that is up for grabs,

so that people can win it.

Another one that we're gonna have to need

is we have to keep track of the total tickets.

If we don't have the total tickets,

let's actually call this total tickets.

So total tickets.

If we don't have the total tickets,

we won't be able to figure out what number to choose from

for our random status.

So we have to keep track of that.

We wanna make sure that we have a price

for all these tickets.

So if there's a price to pay to get in the lottery,

that's something we're gonna have to keep track of.

We're gonna need two things that are not really obvious,

but we'll need the authority.

So there's gonna be have to be an authority

that can do the commit randomness and choose winner,

otherwise you can't really do much with it.

And then finally, you'll need that randomness account.

This randomness account is just to tie the account

that is required to do the randomness information.

We wanna tie it directly to the token lottery config.

Alright, so we have a basic understanding

of what our instructions are gonna look like.

We're gonna have six different instructions.

We have our config, let's go and start building.

So we're gonna open up VS code

and I'm going to do new terminal.

And as usual, we're gonna do npx create-solana-dapp,

and I'm gonna type it all out.

So this is going to be a lottery.

We're gonna call it Token lottery.

Token lottery.

We'll call it Next.js.

For this project,

we're not actually gonna build a front end.

It'll kind of be like a challenge for you to figure out how to build that front end.

We're gonna do a focus on the smart contract side tailwind.

We're gonna come with the basic stuff

and it'll create.

So I'll see you in a moment.

All right, so we have a project,

let's go ahead and open it.

So we go over here, I'm gonna open token lottery.

There we go. We have our project.

It has an Anchor folder.

Looks like it has everything we need to get started.

Let's close all these warnings

and I'm going to open a new terminal,

and we're gonna do first two things, npm install

just so that we make sure we have everything

that is needed here.

All right, now that's done,

then we're gonna go into the Anchor folder.

Lemme make this bigger for y'all.

We're gonna go into the Anchor folder

and we're gonna do an anchor build just so that we make sure that it sets up everything.

So right there we saw something

that we want to keep track of.

It says warning: anchor version 0.30.0

is not equal to the CLI version.

So as a note, throughout this entire bootcamp,

we're gonna be doing it with the 0.30.1 Anchor version.

So we wanna make sure that we set up everything

to be using that.

Alright, so what we're gonna do

is we're gonna go update that.

So if we go over here into programs.basic Cargo.toml,

you'll see down here it says anchor lang 0.30.0,

update that to .1.

We're also gonna change a couple places.

We're gonna change this to say,

we're gonna say token lottery.

We just wanna make sure that sets everything.

This one's gonna be an underscore.

There we go.

Actually, let's just make everything underscore

just so that it's easy for us to understand and remember.

So I'm gonna update this.

I'm gonna update this.

We're gonna update this folder path name also

just to be...

Token, or token lottery.

The lottery, lottery, okay.

Token lottery.

So let's update all the names.

Let's make sure that it's set here.

So there's gonna be a programs here. There we go.

Update that to token lottery.

And then finally, we gotta go into our lib.rs

and update this to say token lottery.

Okay, so now we can do an anchor build.

So everything completed, everything ran perfectly,

so we can actually continue.

So we have here, we have our greet.

It has a message of GM,

so good morning.

We're gonna go and remove that.

And we're gonna go back over to our specific design diagram that we created.

So here we have initialized config,

so we're gonna build that one first.

So we'll call this instruction initialized config.

We already have the context of initialize,

it's gonna do a bunch of different things.

Right here, we know that we're gonna create an account

because we have to create that specific config account.

We're gonna call it the token lottery account

with all that data that we had over here.

All right. So because we know that,

we know that we're going to have a payer,

it's gonna be a signer.

I'm gonna use lifetimes here.

If you don't know what lifetimes are already,

I would recommend looking 'em up in the Rust book.

They're not that hard.

All right, so this is gonna be account mutable

because we are going to use a pair

and there's gonna be lamports deducted from the account.

And then we're gonna have another account.

So this is gonna be account and I'm gonna enter here.

We're gonna initialize it.

The payer for this will be payer.

I'm just gonna put eight here.

We're gonna update that in just a minute

once we create the account.

Then we're gonna have two different things

'cause it's gonna be PDA.

I'm not gonna do it as config,

I'm going to do it as the token lottery.

And because it is a PDA, we're gonna need a bump.

And so this is gonna be a pub token lottery

and it's gonna be a specific account.

We're gonna call this info token lottery.

And because we're creating it,

you can see Copilot already knows

because we're creating an account,

it's gonna say like, hey, we need the system program.

Let's go add that system program.

Alright, so we have this token lottery,

we have to actually define it now.

So let's go to define it real quick.

We need an account.

It's a pub struct TokenLottery.

You can see here, it's asking us to do a bump,

and we're actually gonna keep that.

It wasn't in our original design, but best practice,

if you're using a PDA, you want to save the bump

in the actual PDA.

This allows you to use less compute later on.

Just make sure that you always save the bump in the PDA

and then use that to derive it later.

Alright, so let's set all this stuff in the TokenLottery.

Before I do anything, I'm going to do the,

what is it called?

Derive the macro for derive InitSpace.

And then because I did that, I can now go the eight bytes

for the anchor discriminator

plus the token lottery INIT SPACE.

This will make sure that it auto generates,

or auto generates the amount of bites that it takes

to store this token lottery on chain.

And I don't have to do some math, I always am for that.

Alright, so let's go ahead and set everything.

So first off, we need a winner.

We're gonna have U64 as the ticket numbers.

Next thing, let's say we need the winner claim.

So this is gonna be winner claimed.

This is gonna be a bool.

Next one's gonna be start time.

We're gonna do this in slots, which is a U64.

We need the end time.

We needed a lottery pot amount.

So pub lottery pot amount.

This is gonna be U64.

The next one we have is the, let's see the total tickets.

So pub total tickets, it's gonna be U64 as well.

We have the ticket price and then we have an authority.

So this is gonna be pub authority.

This is going to be a, let's see, it's gonna be a Pubkey.

Let's remove that extra space right here.

And then finally we have the randomness account,

which is gonna be another Pubkey.

So randomness account, randomness account,

not randomness account.

There we go. Alright, so we have this big struct,

we should have everything we need to get started.

Let us start filling this in.

I'm gonna do a quick anchor build just to make sure

that we wrote the right code,

and we don't have any typos anywhere.

Alright, everything compiled, it's great to hear.

All right, so what we're gonna do

is we're gonna start filling out this configuration.

So in order to do that, we just start writing all of them.

So we're gonna remove this underscore here

because we're actually using the variable.

So accounts. we're gonna do the token lottery

equals, we wanna make the bump

because we always need to remember to set the bump.

We're gonna do context.bumps.token lottery.

This is basically just gonna set it in the account

for the bump that was used to derive this token lottery.

Next up we have the start, end date and price.

There's a start, end time and price, right?

Those are actual parameters that we need to pass in.

So let's go ahead and pass those in.

So we have start of a U64 and U64,

and price also as a U64.

So we have this, we're gonna have context.accounts.starttime

is start, end time is end.

Set the ticket price.

We want to set the authority,

so the authority will be the payer.

Then we need the lottery pot account amount to be zero.

It actually sets it as zero by default,

but we're gonna set it just to be verbose.

We want the randomness account to not be set yet.

We're gonna do that whenever we commit the randomness.

So .randomness account equals Pubkey default.

So it's gonna be a default pubkey.

Can't derive, that's fine.

And then finally, we're going to do

context.accounts.token lottery winner equals,

well, not the winner.

We want the winner claimed equal to false.

Let's do the winner...

Let's change just the winner chosen

just to be a little bit more understanding what we're doing

'cause it's not...

It's false. We haven't chosen it yet.

We're waiting to choose it before we continue.

All right, so we set everything, let's go ahead and test it.

All right, compiled, let's go over to the anchor tests.

We're gonna update this

to say token lottery instead of basic.

So token lottery.spec.ts.

And I'm gonna call this the token lottery

and should init config, should init config.

So then a couple things here,

what we see is this is trying to import basic.

We need to import, instead of basic,

we're gonna have to import the token lottery.

So this is gonna open token lottery instead.

And that's gonna be token lottery.

This is so that we can use the program.

And then I'm just gonna copy and paste this right here.

This is so that we're using the program from token lottery.

Now, note this might be different than other tests

that we've done throughout this bootcamp.

We're gonna use the standard anchor test instead of bankrun

for this one because we're gonna do a lot of fun key things

with accounts and programs.

A fun challenge would be to figure out how to do this

with a bankrun on your own.

But ultimately, like it's only a few lines

of code difference to get set up.

All right, so I'm going to remove these.

One other thing that I'm gonna do

is I'm gonna create a provider

equals anchor.AnchorProvider.env.

And then I'm gonna set the provider here.

This is so that I can grab the wallet,

which I'm now going to set as the wallet down here.

So we'll set also a wallet.

Const wallet equals,

this is gonna be provider.wallet as anchorWallet.

Alright, so we have the wallet,

we have the provider, we have the program,

now we can actually do things with the program.

So let's go ahead and do it right here.

So I'm gonna do const.

We're always going to do the constructions instructions

first because it's simpler to understand what's going on.

So this is the init configuration instruction

equals await program.methods.initialized config.

And you can see here, has an error.

Let's go figure out why it's missing.

So it's because we don't have that start time,

we don't have that end time

and we don't have the, what's the last one that we needed?

So start, end, and price.

So let's go ahead and set those.

It's gonna be U64, or sorry, new anchor.BN

because that's how you set the price or a U64,

we're gonna do a few things.

I'm gonna make the start of zero.

We're gonna go grab a Unix time from online

that's gonna be really high just so we can test this.

So just grab that real quick.

Boom, that's really high.

Let's make it even higher.

So it's way in the future.

And the price.

Let's make the price 10,000.

This is 10,000 lamports.

All right, so we have the instruction,

now we have to actually create the transaction.

So const tx equals new anchor.web3

Transaction.add initialize config instruction.

Let's see what happened with it.

Let's see, argument type of pay readable.

Okay, so we're just missing something.

That's fine, we can do that.

So this is just because it's missing some of the information

needed to complete.

So let's go add all of that.

All right, so the things that we need for this transaction

is we're gonna need some things from the blockhash.

So we'll do a blockhashWithContext equals await

provider.connection. we're gonna use getLatestBlockhash.

There we go.

And now we just need to set everything right here.

So the feePayer, we're gonna set the feePayer as my wallet.

We're gonna set, let's see, we're gonna set the blockhash

as the blockhashWithContext.

And then finally we're gonna need the,

it's the lastValidBlockHeight, which is going to be

blockhashWithContext.lastValidBlockHeight. There we go.

And let's see, all the errors didn't disappear.

All right, so we're just still missing something.

This is because, let's see, this is missing an ad.

Let's see, I think it's because we don't have

the right information to run this.

So let's go back up here.

So it's program methods, initializeConfig. Perfect.

Oh, it's not an instruction.

This is a builder.

So we have to actually set as .instruction.

There we go. Everything's happy.

Yep, we got the transaction.

This is our transaction signature,

and now we just have to send it.

So what we're gonna do

is we're gonna do const signature equals await provider.web.

What connection? Actually, we'll do it with anchor.

So anchor.web3.sendAndConfirmTransaction.

This will need a connection.

So we're gonna do the provider.connection.

We need the actual transaction

and we need the actual payer as well.

So wallet.payer. There we go.

I'm gonna add a semicolon just for...

I'm making it look nice.

And then we just do a console.log,

your transaction signature.

I'm gonna go ahead and pull this one out

'cause we don't need that until afterwards.

Let's run it.

So anchor test.

One thing I'm gonna do is I'm going to run

a separate local test validator.

So we have, let's actually remove that to terminal.

I want a specific type of terminal.

I want my zsh, not just a plain bash terminal.

So separately, we're going to set up a local test validator.

So if I run local-solana-test-validator, I'll run my test validator.

And then here I'm gonna do:

anchor test-- skip local-validator.

This is just so I don't have to wait

for the startup every single time.

Also, we're gonna do some special things

with the local validator.

This will make it nicer. So let's run it.

Alright, so it failed, let's go see why it failed.

Should init config.

Program error 1004, so that's a custom program error.

This is usually...

I'm probably just missing something.

Let's actually, we're gonna do a trick here.

We're going to add a skip pre-flight tx

so we can actually read the full error.

So we're gonna skip it, gonna run that thing again.

And then we're gonna go check that errors.

All right, so let's grab that transaction signature.

There we go. We're gonna go over here.

We're gonna go to our explorer.

Make sure that you set it to local.

Go look at that transaction signature.

And you can see here, DeclaredProgramIdMismatch. Perfect.

So that's a better error.

Let's go figure out what that means.

This is, because if you go over into lib.rs,

you can see that it says Hvx.

However, when we deployed it now is ESFW,

so let's go update that.

And I believe you have to do it in one other place.

Let's go find it.

I believe it's in the cargo,

or it's in the anchor.toml right here.

So that also needs to be ESfw.

Alright, let's run it.

Let's do an anchor build with it real quick

and then we'll do an anchor test.

And that way it will start up

with all of the right program ids.

Anchor test and deploy again,

we'll deploy to that same program id and then pass.

So we can see here if we can go grab

that signature real quick and I'm gonna go paste it in here.

There you go. Success.

And when we go all the way down,

we can see initializeConfig, returned success,

and then it took some compute units.

Fantastic. It initialized our config,

it created our account,

now we can move on to the next step.

Alright, so we've initialized our config.

The second thing that we had here,

if we remember from our design

is we have the initialized lottery.

So there's a number of different things

that this initialized lottery is gonna do.

I'm gonna move this initialized config over here,

and just so we can draw up all the different things

that this lottery is gonna do 'cause it's gonna do a lot

and it's gonna be quite a lot of work as well.

All right, so this initialized lottery, right?

So we're gonna do initialize lottery.

It's gonna do quite a few things.

The first thing it's gonna do is it's gonna create a collection that is owned by the program.

This is because we want the program to have full control

of minting more tickets.

We don't wanna have anybody else have that control

other than the program itself.

It's a separation of concerns,

security thing. That's fine.

So create a collection that's owned by the program.

This is a lot of work.

And in that, this will create mint.

So create collection,

create something called a master edition,

so that you can mint more tokens.

Or sorry, this is not a master edition,

it's a metadata account.

So metadata account.

And then we're gonna also create the,

or verify the collection.

All right. And if you go over here,

so if we go over to the token metadata docs,

we can see here that it's very important to check that this verified thing,

let's see if I can make this a little bit bigger.

All right, so this is kind of what we're creating here.

We're gonna create a verified NFT,

which includes the mint account and the metadata account.

We're gonna have that collection NFT,

which is a separate NFT that will have a collection,

some details, and have a mint account.

And then we're gonna verify it,

which sets this collection metadata in the metadata account

to the key equal to the metadata

as well as verified equal true.

So this is what we're basically gonna build

is these four accounts right here.

Let's go ahead and start building them.

All right. So first thing we're gonna do

is we're going to create a new instruction.

So pub function, initialize lottery.

This is gonna have a new context.

We're gonna do Context InitializeLottery.

This is gonna return a result.

I'm gonna remove all this extra code.

And now we have to actually build out

the initialized lottery context.

All right. So let's go ahead

and throw it all the way right after initialized,

we're gonna set it right here.

I'm going to make my terminal a bit smaller.

So it's gonna be a derive accounts, so.

And then pub struct initialized lottery,

and it's gonna have a bunch of different things.

So one thing it's gonna need is it's going to need

the, let's see, it's gonna need a signer.

So let's do account mutable, pub payer signer.

And then we know that we're gonna be creating accounts.

So let's already create that as well.

So system, program, program system.

All right, so we have this initialized lottery.

I'm gonna go ahead and do this real quick.

We've started writing the code for the lottery thing.

So let us compile it and make sure

that we didn't make any mistakes in the meantime.

Okay, so everything worked just fine.

Because we're gonna be used token accounts,

we wanna make sure that we add the specific cargo crate

or the specific crate for anchor-spl.

So anchor-spl.

This is actually, it's a cargo add.

What is it called? Anchor.

Let's go grab the docs real quick.

It's always good to have the anchor lang docs available.

So this would be SPL, was is it?

Anchor-spl.

All right, so let's go grab this anchor-spl thing.

So there we go, anchor-spl.

So we need to add this right here.

There we go, it's adding.

Okay, it looks like Rust analyzers are still catching up.

Another thing is we're gonna have to make sure

that we add something right here

of anchor-lang/ or anchor-spl/idl-build,

just so that we make sure

that we have the right IDL stuff for it.

Right here we have anchor-spl.

We want to make sure that we have the features of metadata, so I'm gonna add it right here.

So this is version equals 0.30.1/features equals metadata,

I believe is what it's called.

There we go.

And I'm gonna do an anchor builder one more time

just to make sure everything is set up correctly.

Alright, so it's because, let's see,

I missed something here.

Oh, I missed quotes.

That's fine. I'm okay with that.

All right, so compiled,

that means we added everything correctly.

I always like to do that just to make sure

that everything is right.

Let's continue with creating this initialized lottery.

So we have the token, we have the assistant program,

the payer, I just spoke about what I wanted to add as well,

we wanna make sure that we add the token program

that we're gonna use.

So pub token program.

This is going to be a, what is it?

An interface program, because we wanna make sure

that we allow any type of token program to exist.

So interface of info, I believe it's what?

Token interface. Yeah, token interface. There we go.

All right, so we have the right token program.

We're also gonna need the associated token program.

So pub associated token program.

This is gonna be of type program info,

not token interface, but associated token.

There we go.

We're also gonna need what's called the token metadata

program because we're gonna be creating those metadata.

So token metadata program,

program, info TokenMetadata.

It's gonna say that we need to add it,

so let's go add it real quick.

We need to make sure that we add it here and save.

Running into an error.

We'll fix it in just a moment. And that's okay.

I was not satisfied with, ID is not satisfied.

All right, we'll fix that in just a moment.

I'm just gonna make this a little bit prettier because otherwise it's gonna be a mess in the long run.

Oh, I think this is the wrong...

TokenMetadata is the issue.

All right, so let's add all that.

This metadata that we need is actually

from a different crate.

Let's see if it knows it.

otherwise we're gonna have to add it individually.

Oh, so it's not actually TokenMetadata. It's metadata.

And this is under anchor-spl Metadata, Metadata. Perfect.

Alright, so everything is happy there.

Next up, we need to actually start creating

all of our different accounts.

So these are our collection mint, our metadata account,

our actual collection token account,

so that we can have it on chain.

And then finally our master edition account.

So let's start off with some of them.

So first off, we're gonna do the collection.

We're gonna have to do the collection mint.

So this is gonna be an account.

This is gonna be init

because we're gonna initialize something,

payer equals payer, so that we have something paying

for this thing.

Since this is a mint, we're gonna have mint decimals of zero

because it is gonna be a collection mint.

This is gonna be mint authority.

I'm gonna set to payer,

but we're gonna change it in just a moment.

We're gonna also change this one to be something else

in a moment as well.

We also need the seeds.

We're gonna make it a PDA

just so that we can easily derive it later.

We're gonna call this collection_mint as ref.

And because you have seeds, you must have a bump.

Now unfortunately, because this PDA is a mint account,

we can't actually store the bump inside the mint

'cause it doesn't have a space for it. That's okay.

We'll be fine with that.

This is gonna be a type interface account

because it could be a different type of token account.

Info mint. There we go.

And this mint, let's see,

this mint should be the interface mint.

All right, save.

So that is our collection mint.

The next two are...

Let's do the collection token account first.

So the collection token account is an account.

It's gonna be...

We're gonna initialize it.

We're gonna have a payer. That's perfect.

Oh, it did a whole bunch of things.

I don't want all of this here, so I'm gonna remove all of it

and we will keep going through it in just a moment.

All right, so we have the token mint equals

the collection mint and then token authority.

We're gonna have the authority to be at itself.

So we'll leave it for payer, we'll update it.

So pub collection token account is a type interface account

because it can be either type of account/info

TokenAccount and it's gonna be token interface.

Perfect. So we're gonna update a few things here.

I don't want the payer to be the authority

of the collection.

We want the PDA to be authority.

Same with the collection token account,

we don't want the payer to be authority.

So we're gonna make the collection TokenAccount

the authority of itself.

Alright, so we have those. That looks fine.

Just to make sure that we're doing all right,

I'm gonna do an anchor build.

Everything compiled fine, we can keep going.

All right. So the next two things.

We're gonna do something that's not normally best practice,

but it's okay in our case.

You'll learn in the security module,

if you went through it already

why it's not best practice, but it's okay.

So we're gonna use two different accounts.

The first one is going to be the metadata account.

So metadata. And this is gonna...

We're gonna call it metadata, an UncheckedAccount.

And the reason being is it's gonna be checked and created

by the actual metadata program.

Another one is we're gonna need the master edition.

Also, same thing.

It's gonna be checked by the metadata program.

Because you do these two things,

what's gonna happen is when I run this,

it's gonna say error, error, warning,

you have an unchecked account.

It's always good of an error to get,

because you need to know what you're doing

when you're using these.

You can go check the code

for the token metadata program right here.

You can read it, you can see

that it's actually gonna be checking these accounts.

And since we're CPIng, we should be okay.

So let's do an anchor build and see that error.

All right, as expected, safety checks failed.

Didn't work, we didn't want it to work. Great.

So we're gonna have to add some checks

to say like these are checked by the token meta program.

So this account is checked by the meta data smart contract.

And I'm gonna copy this and put it in two places.

All right, so we have these.

Now, there's one more thing I wanna do

is that these accounts can actually be derived.

They don't have to be just mutable.

We're gonna actually set all the seeds,

and stuff here so that we don't have to derive them ourself.

If we derive them ourselves, it's more work on us.

Let's just make it auto derived every single time.

All right, so for this one,

we have to add those seeds.

We're gonna add empty seeds real quick, needs a bump.

And then finally it will need to specify the program

that it is getting or driving these from.

So this would be seeds program

equals the token metadata program.

Alright, and the seeds for this one are as follows.

First, it's metadata,

then it's going to be the metadata program.

So token metadata program.

Next step is the mint.

So we're gonna have to do the collection mint as ref.

And let's see, it's got an error.

Let's figure out what it is.

It's just missing a comma somewhere.

Oh, it's missing a comma right above it. That's fine.

All right. All right. So this needs to be the key.

Yep, key. That's fine. Save it.

And there we go.

So we have these as the seeds here

and we're gonna copy and paste this real quick.

So I'm gonna copy and paste this account real quick

'cause it is very similar.

And instead of, let me just do some enters here

so that it's easier to read.

So token, metadata program collection, mint.

And the final one for this one

is we have the token metadata program, collection.

And the last one is not actually empty,

but it's going to be addition.

And I need a comma up ahead.

And let's see what, oh, I had a dot. That's fine.

So if you're ever interested in how to derive this,

you can actually drive it just based off

of the smart contract itself.

So we can go to the code, we can see here,

here's the seeds for metadata.

It's that prefix that we have the program id,

and mint info, everything's there.

You can also do the code diving for this one as well.

This is for the master edition.

You can just go to the mpl token metadata program,

it's on GitHub.

You can go look through it all.

But we have everything here.

We have our checks, so CHECK,

let's make sure we have those,

otherwise we're gonna get another error.

Let's now go ahead and anchor build it.

Alright, so it's saying that token metadata is not there.

Let's see, it's right here.

So token metadata program,

I probably just said it incorrectly and that is fine.

Token metadata program,

token metadata program, token metadata program.

Let's see, what is not in scope.

Let's find it.

So one thing I notice is it's should be .key

because otherwise it won't know to use the key.

It'll just use some, I guess some bites or something.

Let's go and build it and see if that fixed it.

Yeah, that'll fix.

It just needed to be .key.

Yeah, so it's just because it cannot find it

'cause it needed to be a key instead.

Right, so we have everything, it builds.

Now we can actually start writing the code

for initialized lottery.

So there's gonna be a whole ton of code

in this specific function.

We're gonna go through it line by line.

Alright, so the first thing

is we're gonna do a bunch of things with CPI

and they're gonna be signer seeds.

We're gonna need signer seeds.

So we're gonna do let signer seeds,

we're gonna do equals,

it's gonna be a type of reference, reference,

reference U8.

This is how our signer seeds look like.

We're gonna go backwards there. There we go.

Equals reference, oops reference.

We're gonna have to add our collection mint.

This is because we're going to use the collection mint

to do a bunch of things as ref.

And then finally reference.

Let's change this from _context to regular context.

Context.bumps.collection mint.

So now we will be able to sign with the PDA,

that is the collection mint account

to do a bunch of things which we'll need in just a moment.

So first thing we need to be able...

I'm gonna create a message

that's gonna be Creating Mint Account

'cause that's the first step that we're gonna do

as referenced when we were doing (indinstinct) draw.

We have to create the mint, create the collection,

create the metadata account and verify the collection.

All right, so let's go ahead and create that mint.

This is gonna be something that you've probably done before.

Mint to, this needs a CpiContext.

We're gonna do it. New with signer.

New with signer. There we go.

I'm gonna go ahead and put a question mark

there 'cause we need it.

We'll need it in just a moment.

A new signer. We have to provide the program.

So it's gonna be the token program:

context.accounts.token program.to account info.

The next thing we'll need is we're gonna use MintTo.

This is just kinda to help us out

with creating the accounts.

There we go.

You can see here, it already kind of tells us

all the different things we need.

Some of these things don't actually exist,

but we need to make sure the mint is the collection mint, is going to mint to that collection token account

and the authority is not there,

but it's gonna be the collection mint as well.

As we remember, it's the collection mint.

All right, we're gonna make sure that we set it

with the signer seeds and we need to mint one.

So we're minting one to our token. Let's see.

This is saying that I can't find it.

Let's go ahead and add it real quick.

Import mint two.

Yeah, sure, let's do it.

Import from token, yes.

All right, it's saved.

Let's do an anchor build,

and make sure we did everything right.

So one thing I realized is that this mint to

is coming from, let's see,

it's coming from anchor-spl.

We actually need to make sure

that we're using a different mint two.

So we need to import the...

Let's go backwards a little bit.

We actually need to import a different mint to.

And this one's gonna be a token metadata one.

So it's gonna be anchor-spl,

it's gonna be, let's see, not token metadata,

but token interface.

Interface, and this is gonna be mint_to.

Might have to write it myself and that's okay.

All right. So we need to add it here.

Token interface. It's right here.

We need to add the mint_to and MintTo.

There we go.

We need to make sure that this is using the right thing.

Remove that. All right, so everything should work.

Let's just do an anchor build just to make sure

that we did it correct.

Alright, so we have the mintTo,

it's minting a single token to our token account.

Next thing we're gonna do is we're gonna create

the metadata account for that token.

So let's create those metadata accounts.

So message, just to make sure

that we have all the message stuff

for creating a metadata account.

We'll remove these later, that's fine.

So this is gonna be create metadata accounts v3.

You can see it's already kind of knows what I wanna do.

I'll leave this here.

We're gonna fill this out.

It's a lot of things we need to fill out here.

All right, so first is that context.

This is gonna use a new context.

So we need a CPI context.

First one. This one's gonna be new with signer

'cause we have to sign with that.

Keep that collection mint account again,

you can see it already says your program,

what accounts are you doing and your signer seeds.

Thankfully those signer seeds are the same ones as earlier,

so we don't have to worry about that one.

The program is going to be

that context.accounts.token metadata program.

to account info. Perfect.

We're gonna use a fancy helper for us.

We're gonna do CreateMetadataAccountsV3.

This is just to help us out,

and you can see there's a lot of different things here.

Alright, so let's start filling this in.

It's got a bunch of red lines.

Let's make sure that we fill it in.

All right, so the metadata

is gonna be that metadata account, that works.

The mint is that collection mint.

The mint authority is itself, perfect.

Payer, the one who's gonna actually pay for this

is the payer, great.

There's the updated authority.

We wanna make sure that's not the payer.

We wanna make that the collection mint.

So let's go and paste that over here.

System program, a system program.

You can see it requires rent.

We didn't actually include rent.

So let's go ahead to go add that real quick.

So going back down to our context right here, we're gonna have to go add a rent account.

So pub rent Sysvare info Rent.

And we'll just update this up here to b

instead of the system program

that was autogenerated, rent.

All right and then this should be everything.

Let's see what it's missing.

Expected accounts V3 lifetime.

What is this?

Expected found mismatched...

Expected pub key got account info,

so it's missing something.

Let's figure that out.

So I figured it out.

This up here, just the auto import did not work as expected.

It's grabbing it from the wrong import here.

So you can see it's doing import

of anchor mpl token metadata.

That's completely different than what I actually wanted.

It's actually there is a create metadata account V3

right here, should be anyways.

So I figured out here this import is using the wrong create metadata accounts

is using it from mpl token metadata.

So it's an easy fix.

We should be able to just update this here.

It's also because I did it, it's under account.

It needs to be accounts V3.

So we'll just remove this right here. Save it.

It's not being used, you can see

and I can go back down here, update this to say accounts

and then it should be happy.

Alright, cool. All right, so this is a reference

to signer seeds.

We make sure always reference it.

And then we have to create the data as mutable,

updated authority assigner and the collection details.

So we're almost there.

All right, so next one, what we're gonna do

is we're going to create that data.

So this is gonna use a struct called DataV2.

We're just gonna have a bunch of things in it.

See it already tries to set 'em all,

I'm going to set them individually.

I'm gonna go ahead and skip to the bottom

and start doing all these. So true.

We wanna make sure that it is mutable,

so that we can change it later.

The update authority will be yes,

the updated authority is the current signer.

And then we're gonna do some collection details.

Details. V1 of size zero.

There we go.

All right, so we should just have to fill out

this DataV2 instruction now.

So it needs a name.

We're going to create these names in just a moment.

We're gonna just for now...

We're just have an empty one.

We'll need a symbol.

We're gonna need a uri,

and then we'll need a seller fee.

Basics points. We're gonna set a zero

'cause we don't care about this in this collection.

We're gonna have a creators array

or a creators struct, I should say.

Well, it's an array of some vec.

We have to create a creator.

We don't really care about this,

but sometimes you might care.

This is for like profit share,

or sharing of fees amongst the creators.

We don't actually care in our case.

So that's all right.

Context.accounts.collection mint.key.

We're gonna have a verified as false

because it's not verified yet.

And then share of a hundred.

We don't really care about all this.

We're not gonna use it.

Unfortunately, you have to include it,

but we don't care about it at all.

The next thing is you have a collection

is none at the current time.

We're gonna change this later and then uses, none.

This is also true.

Okay, so the thing that we're missing here

is it says it doesn't know what creator is.

Let's go import it real quick.

It's trying to import it from there.

We don't want it to create from there.

So let's make sure that we're importing it from one place

and believe it is just under metadata.

So let's make this nice and set up here.

So let's see if it exists under there. No.

All right, I'm gonna be right back.

We're gonna check where it exists.

All right, so where it exists

is it's under mpl token metadata, types creator,

boom, we have the creator.

It's being used. Unnecessary braces.

Yeah, it's not necessary

because we're only using one type as of right now.

All right, cool.

And that should have everything there.

The next thing that we're gonna have to do

is we have this name symbol and uri.

Let's go. Adam is constants up here.

So we can use this throughout the program as needed.

So we're gonna do a constant,

this is gonna be pub const,

and we're gonna do the name first.

This is gonna be of type and string.

It's gonna be...

We're gonna call it Token Lottery Ticket number.

And you'll notice that I did it with no ticket number.

This is because we're gonna add the number

for the ticket later.

The symbol. We're gonna call this, TLT.

Sure, why not?

The constant. This is gonna be the ticket constant

so pub URI and string.

It's gonna try to do some arweave address.

We're just gonna go grab something from the internet.

So let's go grab it real quick.

So let's just go ticket and we'll grab a ticket here.

So admit, well this is a nice ticket.

And go and copy the URL image address

and put it right here.

Oh, that's a big address.

Ah, we'll figure out that.

Breaks it later. That's fine.

All right, so we have all those things.

Let's just go set them right over here.

So this is gonna be name,

I guess it has to be to string.

This is gonna be the SYMBOL to string.

And then finally the URI to string.

All right, let's go and build it, see if it works.

All right, so compiled, we must have done everything right.

The next thing we have to do if following our things

is we have to create the actual master edition.

So let's go create that master edition.

So I'm gonna do message, we'll do creating master look

and do exactly what we need to do next.

We're gonna have to use crate master edition v3.

It's gonna need a context and a max supply.

This max supply will be a sum zero

because we're not going to be doing more

than just this account.

But let's create that context real quick.

So context. Whoops. There we go.

I'm just going to pull it out so it looks nicer.

All right, so another context.

So we need a CpiContext.

This one, once again, new with signer.

This is because we have to sign for everything.

We're gonna do a signer with that collection mint.

And so we're gonna pull this out.

We know that we already have the signer seeds,

so we're good there.

This is gonna be

that context accounts.token metadata program

to account info.

And then this accounts

is going to be a CreateMasterEditionV3 right here.

And we're gonna need quite a bit of accounts.

So first one we'll need is that payer,

is gonna be that payer info.

Then we're gonna need the mint.

This mint is gonna be that collection mint.

So thankfully we already have that.

Next one is gonna be the addition.

This is gonna be that master edition account

that we've passed in.

The mint authority

is gonna be the collection mint once again,

the update authority collection, mint.

That collection mint has a lot of power and that's okay.

Metadata is gonna be that metadata account.

The token program is gonna be the one that you pass in,

the system program for creating the account.

And then finally, rent.

I'm hoping in the future

that this rent's not required anymore

because you actually don't need it.

All right, cool.

So we've created the master edition,

let's anchor build, make sure we did it right.

All right, so we did that.

And the final thing, if you remember

is we have to verify that collection.

So let's go ahead and verify that collection.

So we're gonna do a message,

Copilot did not get it right this time.

Verifying collection.

We'll remove all these messages later.

It's just gonna be helpful for us to understand

what's going on.

So this is gonna be a sign metadata function.

It's gonna have a context.

So let's add that real quick.

I'm going to go ahead, go down here, finish it out.

All right. So CpiContext.

Once again, as expected, new with signer,

because we have to sign with the collection mint PDA.

That collection mint PDA

is gonna be used a lot throughout this whole thing to sign.

All right, so program is that token metadata program.

So context.accounts.token metadata program

to account info.

The accounts, we're gonna use a nice little struct

for this SignMetadata.

It doesn't have all these things that it thinks it has.

That's okay. The things it does have is it has a creator, which is that collection mint account as we set up here.

So we set up here to the...

The creator was the collection mint account.

And then we're gonna have the metadata account.

All right, it was happy.

Looks like just making sure my braces all line up. Yep.

I'm going remove this new line 'cause we don't need it.

All right, let's build it

and see if we did everything correct.

All right, so we have everything set up.

Let's go build it out.

I'm gonna add this into this initialize.

We're just gonna make this a general initialized test.

It's not really a test because we're not asserting anything.

That's okay. We just wanna make sure are things working.

We can assert it ourselves.

If you want to write good code,

I would recommend just asserting it

if you want to on your own.

So I'm gonna do it here as well.

So we're gonna do const.

This is gonna be at initLotteryIx.

It's gonna have some accounts.

So this is not everything.

So we'll do .accounts.

It should only have one account, if I remember correctly,

because of what we can do is we can actually check

what the accounts are required.

We can see the payers myself, we already know who that is.

The collection mint, we already know who that is.

Collection token account, we already know who that is.

The metadata account should be derived from these seeds.

Same with the master edition.

What is not given is this token program

because it's an interface,

we have to tell what token program to give.

So this should just be token program,

and we're gonna use the...

It's gonna be TOKEN PROGRAM Id.

We can see it's get grabbed from...

We don't wanna grab it from there,

we wanna just import it. There we go.

Let's make sure it imported from the right place, from spl-token.

If you don't...

I don't remember importing this,

but if it's imported, great.

If it's not, we'll do it in just a moment.

So this is the IX for that,

That's all we need, I believe, for that.

So once again, we'll just create this instruction

or this transaction.

So const, this is gonna be initLotteryTx

equals new anchor.web3, add LotteryTx.

We should be able to use the same information from earlier.

I don't expect time to have passed dramatically.

So our blockhash should still be valid. There we go.

And then finally, let's just do the same thing,

we're gonna go grab this real quick.

This should be a initLotterySignature.

InitLotterySignature.

I'm going to enter so that it looks a little bit nicer.

I'm gonna take out the skipPreflight

'cause we don't need it,

unless something doesn't make sense.

Gonna make this pretty as well.

Alright, so we have provider connection.

We need to make sure that it's using

the lottery transaction.

We have the payer as the wallet.payer.

Let us also console log it.

So console.log, your initLotterySignature

and let's run it.

So one thing you're gonna note

is that if I run this right now without doing anything,

it's going to try to initialize all of those accounts again.

The lottery token, lottery config,

a whole bunch of accounts.

So I'm gonna go ahead and restart the validator with -r.

This resets the state so that we can make sure

that we understand what the state

that we're going to start out with, right?

So let us anchor test it.

Anchor test, skip local validator.

All right, so it failed.

Let's figure out what we are missing to make it work.

So we got a specific error code.

This is on the ininitLottery transaction.

So let us do that same trick as earlier.

We'll do the skipPreflight,

so we can see what is the issue.

So I'm gonna reset my test validator again

and run that test.

All right, so we have a transaction

with a custom signature, a custom error.

We can see it failed on that initLottery.

Let's go see what that error is.

The explorer is, see it failed with invalid program execute,

program account is not executable.

Alright, so I know what this issue is.

This is because we don't have the token metadata program

on local. So if we go to this program,

we can see this account right here.

Account does not exist.

We actually have to go grab that from our mainnet.

So if you go to Solana Cookbook,

you can see here that there's going to be some information

on how to dump programs.

It's program- or Solana program dump-um,

your program id and the .so file.

So we're gonna do that once on our own.

And then I'm going to show you a script

that will get you all the information that we need.

So Solana program dump -um,

this is for that specific token metadata program,

which is...

Let's go grab it from the transaction right here.

And this is gonna be put into metadata.so,

boom, we have the metadata.so

and now if we wanna load it, we have to do bpf program.

We have to set up...

Let's just make sure verify with the code,

bpf program, program

and the SO file.

So metadata.so, reset.

Now if we go over to look at this account, it should exist.

Boom, it exists. Perfect.

And it is an executable.

All right, so let's run that same test again, and see if it works this time.

Alright, so it passed.

Let's go grab that initLotterySignature and look at it.

Okay, so here's the instruction.

We can see a whole bunch of accounts

that we've initialized right here.

We can see the token balance of one

because we have one token in that new token account.

And then there's an instruction,

create account, initialize the mint

and create another account

and initialize the account, we Mint To it.

And then here's our big instruction for, let's see,

this is the...

Looks like this is one of the metadata program accounts.

And then here's another one.

So one of these is gonna be the master edition.

One of 'em is gonna be the metadata account.

And then we set all the authorities.

And then this final one is the verify.

So this verify right here.

So you can see all of our code right here, setting up, creating, it went through and built the whole thing,

signed the verify.

And our collection is verified. So fantastic.

Another thing that we can do,

and let's see if we can pull it, is that this...

Let's see if we can pull the account.

So one of these accounts is going to be our token account.

Let's see. So this is our writeable one.

We can cheat this.

Where's the token account?

Here we go. Address token.

Let's go look at that token,

see if it has the little logo.

So my fear is realized that random image

that we pulled is not showing up here

'cause it had something weird with the URL. That's okay.

But we can see the Token Lottery Ticket right here.

That's the master edition has that same TLT.

So the URI is actually something else.

You can go look at that in the token metadata account.

So let's go look at what the URI is supposed to be.

There we go.

So it's a specific format.

So metadata format.

So if you go look at these JSONs,

it's actually supposed to be a JSON URI,

which includes the image.

We're not gonna do that real quick.

You can go based off of the the previous collection module

and just learn it from there.

I'm gonna go ahead and pull one that I have.

So we'll go grab that from Phantom real quick.

Alright, so the image wasn't generated

because it's actually the URI

is supposed to be a JSON file.

I just grabbed an example one,

we're not gonna create one real quick,

but this is an example one of name, test token,

symbol test, description your test.

Your image and here's your actual image.

So that's why it's not showing up.

I'm gonna go ahead and generate this on GitHub real quick.

So we're just gonna create this on GitHub under the developer bootcamp and I'll be right back.

Alright, so I've created this under

the project eight token lottery.

It's a metadata JSON, it has that image

that I grabbed from the internet.

You can use the same URL.

So what you do is you go click on raw, copy and paste that,

go into here, update your URI to that specific URL.

And then if you go look at now this transaction

or this token, you can see the token has a ticket.

Token Lottery Ticket #TLT.

Right, so now we can continue.

So we've now initialized the lottery,

the next thing that we want to do

is we want to actually buy tickets.

So let's go in and buy those tickets.

I'm gonna just make this,

should test token lottery just so that we can do

the whole thing in one.

This is not necessarily a good test,

but it's just so that we don't have to worry about it every time we restart.

All right, so we'll get back to this.

Let's go to lib.rs,

and start building out that new instruction.

So we've initialized the config, we initialized the lottery.

I'm gonna go and minimize all these,

so it doesn't get in our way.

The next one that we have,

if we go back to our design is buy ticket.

So let us do that, buy ticket instruction.

We have to go afterwards.

Let's go at the end of this.

All right, so pub fund buy ticket.

You can see it already kind of gen, whoops.

Generate some code for us.

I'm gonna minimize this again.

There we go. Buy ticket.

So it's gonna have a buy ticket context.

And we're gonna get started.

I'm gonna head...

We're gonna have an amount,

I'm just gonna have it buy ticket, one ticket at a time just so that we don't have to worry about this.

We can do the amounts on the front end and that's fine.

All right, so we have the buy ticket.

Now we need to create the accounts for it.

So let's go down here and create those accounts.

So it's derive accounts, pub struct, contact or BuyTicket.

That's our contact.

It already tried to generate a bunch of things for us.

We're gonna remove all of them.

The only thing that we we know we're gonna need so far

as we need the signer 'cause we need someone to pay for it.

And then we're gonna need the pub system program

because we're gonna create a new ticket.

All right, so the next thing that we need

is we're going to need a new mint for this new ticket.

So let's initialize a new mint.

So we're gonna do collection, or sorry account.

We're gonna have init again,

payer equals payer as usual.

We're gonna create this as a new seed.

It's gonna be the specific token lottery number that we're gonna use it.

Which actually in order to do that we need to do

the lottery as well.

So let's grab that token lottery real quick.

This is going to be immutable.

It's gonna be mutable, not immutable

because we're gonna be updating the ticket count within it.

It's gonna have those seeds.

Those seeds are going to be b token lottery.

And then the bump is going to equal token lottery.bump.

This is so that I can easily derive it.

So pub token lottery token lottery info.

Alright, so now we have seeds.

The seeds for this one is gonna be token_lottery.ticket.

It's gonna be the number of tickets.

So, but total tickets.

So it's gonna be the current ticket number,

to.le bytes, little Indian bytes.

That's gonna be bump.

And then we're gonna be initializing.

This is gonna be ticket mint.

It's gonna be once again another interface account of info mint.

So we're missing some code here because it's a mint.

We have to provide all the information for it.

So mint decimals equals zero.

Mint authority equals the...

We're gonna have this as the collection mint.

That collection mint is gonna be very powerful yet again.

Looks like I'm missing a colon. That is fine.

There we go.

Freeze authority is gonna be that collection mint again.

Mint, token program.

This is gonna be that token program.

All right, so you can see here

we're already missing some things.

We are missing the token program, and we're going to...

So it's probably gonna say right here,

yeah, missing token program.

And we're also missing that collection mint.

So let's go ahead and create those real quick.

So pub is gonna be the token program, TokenInterface.

All right, so we need that collection mint account.

This collection mint is going to be...

Account is gonna be mutable

because we're gonna increase the number of tokens on it

while we're gonna use it basically.

The seeds are gonna be the collection mint account again.

So collection mint.as ref and then bump.

Unfortunately we couldn't save it

because it's not our account.

Collection mint, Interface, and our mint.

Alright, so we have that.

Next we're gonna need a few things.

We're going to need a metadata account for our ticket mint.

We're gonna need that same master edition account

that we had earlier.

And then we're gonna need also

the collection metadata account as well.

So let's go ahead and build those out.

All right, so first off,

we have a new account.

The first one is the metadata account.

So seeds are the same as earlier,

so let's go grab those seeds.

Where did they go?

It's an initialized lottery.

It is gonna be right here, not here, right here.

All right, so the seeds are these,

and go ahead and grab all of this code right here.

How much should I grab? I forgot.

There we go.

Okay, seeds collection mint

is not gonna be the collection mint.

It's gonna be that ticket mint.

All right, let's go ahead and grab this,

this is gonna be the metadata account.

So pub, we're gonna say is a ticket metadata.

Once again, it's gonna be created

by the metadata smart contract.

And the other one that we need is that master edition again.

So let's go grab that.

You need this so that you can add a NFT to a collection.

So instead of the collection mint,

it's gonna be that ticket mint again,

this is gonna be the ticket master edition.

Right. So that should be right.

Okay, so it's already telling us need the metadata program

because we're doing stuff metadata program.

So pub, token metadata program, token metadata, perfect.

Okay. And the next thing that we need to do

is since we're buying a ticket, we need a destination,

we need a token account to give for this.

So let's go and put that in here.

This is one thing we're gonna need

is the associated token account

which we'll do in just a moment.

So account, init,

payer equals payer.

We're gonna do the associated token,

mint equals token mint.

Associated authority, this one's gonna be the payer

because we're gonna give it to that person.

And then associated token: token program

equals the token program.

And then we have to create that specific account.

So it's gonna be a pub destination

equals InterfaceAccount of type info,

lifetime info TokenAccount.

Perfect. All right.

And because we're using the associated token program,

we once again need the associated token program set.

So let's set it down here.

Pub, associated token program

associated token.

And if you remember when we did the master edition earlier,

we also needed rent.

So let's add that real quick. Boom.

Alright, so that should be everything for buying tickets.

Now let's actually write the code to buy the tickets.

Go back up here and let's get started.

Before we get started, let's actually do an anchor build

just to make sure we're doing all right.

All right, it passed.

Just had a warning because we have the context.

That's okay. So first thing, I'm gonna get that clock,

so that we can tell if whether or not someone

can actually buy a ticket.

So now we have the clock.

Then we're gonna get the ticket name

just so that we make sure that we set the ticket name.

This is gonna be NAME,

we're gonna do a little bit of string concatenation in Rust.

This is gonna be plus context.accounts.token lottery.

total tickets.tostring. Astring, boom.

And now let's check.

So if clock.slot is less than context.accounts.

token lottery, this is gonna be lottery start time

so that the lottery is not ended,

or clock.slot is greater than the end time

because we don't want anybody buying after it ends.

We wanna throw an error, we'll go at LotteryNotOpen.

You can see here we already have an error code.

So that we need to actually create those error codes.

So let's go down to the bottom and create those error codes.

So the way that you create error codes

in Solana programs or Solana smart contracts

is you use the macro error code.

Pub enum, you can see it actually already knew.

Lottery is not open, it's a message. That's the message.

There's our enum.

Alright, so let's scroll back to our code.

We have the ticket name, we've done the clock slot,

check that someone is eligible to buy the tickets.

Now we actually have to do the buying of tickets.

So if you remember there was a price to pay

whenever you do the ticket.

So we're going to take that money from the buyer

and deposit it into our lottery pot.

So CpiContext, this is gonna be new

'cause we don't need any PDA signatures here.

We're gonna do context.accounts.

this is the system program.to account info.

We're gonna have to do system program::Transfer.

Whoops. There we go.

Alright, so it's gonna be from the payer

to the token lottery.

It's not the authority 'cause that would be myself.

It's gonna be the token lottery.

It's the token lottery account itself.

And the lamports is specifically the token lottery price.

That is not in this section,

it is actually outside of the transfer.

It's down here and it's actually just the lamport amount.

There we go.

Alright, so we've done the transfer.

Next thing we need to do is we need to create the ticket.

So as earlier we're gonna do a lot of the same thing,

we're gonna do let signer seeds.

It's not gonna be token lottery,

it's gonna be the collection mint

'cause we have to use the collection mint authority

to create a bunch of these...

A new ticket off of this collection.

And we have to mint the new token.

So mint to once again,

CpiContext new with signer.

We're gonna just enter a bunch of things real quick.

There we go.

The signer seeds are a reference.

The program is gonna be a context.accounts.token program.

to account info.

Then we're gonna use the MintTo,

the same one as last time.

And we're gonna mint to...

The mint is gonna be, let's just set all that.

It's the ticket mint.

We're gonna mint it to the destination,

and the authority to do that minting

is going to be the collection mint.

Perfect. Alright, so let's see all the errors that we have.

So first off, I realize that this needs to be bumps

because that is the current way that you grab the bump

and see that MintTo instruction.

What is? So I'm missing something in the instruction.

We're missing the amount. That's okay.

Very easy fix. It's gonna be comma one.

There we go. So we minted a single token

to the ticket of the ticket mint.

Next up we're gonna go create the metadata accounts

and create the master edition.

I'm gonna go ahead and grab that from the token lottery

and we'll just update it as needed.

So these two right here.

I'm gonna copy and paste those

just so they'll save us a lot of time.

All right, right here.

And the first one we need to go through is this one,

the CreateMetadataAccounts.

We need to make sure that it's all set up correctly.

So the correct one should be ticket metadata.

The mint is the ticket mint.

The mint authority is the collection mint. That's correct.

The payer will be the the payer.

The update authority is the collection mint.

System program rent, perfect, signer seeds.

Our name, our string.

This is actually gonna be our ticket name

because we updated...

We wanna have our ticket name be the one with the number.

Let's see. Okay, so sellers fine,

address is going to be the, let's see,

this is the creative address.

We don't actually have any creatives for this one,

so I'm gonna go ahead and remove this and put none.

True, true. And then this is going to be none

because it has no collection details.

This is just like, if you remember

from the token metadata information,

the specific mint account has none.

Alright, so we have that one,

next up we have the master edition,

so let's go and start creating it.

So this one, the first thing I know

is this is gonna be the ticket mint.

The master edition is gonna be the ticket master edition.

Once again, also the metadata is ticket metadata.

And let's go down payer.

Yep, the mint is the ticket mint,

ticket master edition.

The mint authority is the collection mint,

update authority is collection mint.

Metadata, ticket metadata,

yep, rent, everything good.

Sum is zero, perfect.

Now we need to actually verify

this as part of the collection.

So set and verify, or set and verify side collection item 'cause we're adding a new collection item.

We're going to go ahead.

This is the authority record.

Remember earlier we set it as none.

So we're gonna set it as none now.

And I'm going to make this a little bit prettier.

Enter, enter, enter.

All right, so we need a new CpiContext.

Once again, it's new with signer.

So new with signer.

Let's see if it did everything. That'd be fun.

So it needed to be a token metadata program. Perfect.

Set and verify size collection item, that is correct.

Now we're missing a bunch of the different items,

so let's just fill that in.

So the first one is the metadata.

This is just gonna be the context.accounts.ticket metadata.

to account info.

Next one is gonna be the collection authority

which collection authority, which is that collection mint.

The payer is going to be the payer that we're used to.

We're going to have an update authority, which is gonna be the collection mint again.

We're gonna have the collection metadata,

context.accounts, do we actually include this?

Did we forget something?

I think we forgot something.

That's okay. Let's go add it.

Two of the things that we...

We actually missed two things.

I just double checked.

We missed the collection metadata

and the collection master edition

and collection metadata.

So we're gonna copy and paste this.

We're gonna change a couple things.

So this is gonna be collection metadata.

And this one down here

is gonna be collection master edition.

And then we just need to change these

to use the collection mint account.

There we go.

Alright, go back up to our code.

So collection metadata should be now collection metadata.

to account info.

The next one, let's say next one,

I believe it is the collection master edition.

So collection master edition,

which is or collection master edition.

And let's make sure I have everything.

So looks like you don't.

Let's see what I'm missing.

I'm missing a...

So whoa, okay, so I'm not importing it.

Let's import it. Let's do this one.

Let's make sure it import in the right place.

One thing I don't really appreciate

is it's putting it all here.

I actually want it to all go up here.

So let's go pull it out, and put it up here.

So create master edition.

Just going to clean this up a little bit.

It's adding it in a place that I'm not exactly fond of.

This is why you don't do auto complete sometimes.

So let's go and copy and paste this whole thing into here and then just update it as needed.

Pull it out.

All right, let's see if it added everything.

Oops, I put a extra character. That's okay.

All right, so where is it?

SetAndVerifyCollection, looks like I'm missing something.

I'm missing collection mint and collection metadata.

Ah, I added an extra thing.

And then we need the collection mint.

So collection mint,

as expected collection mint.

And it's all happy.

All right, so we've done all this.

Now that we've created the ticket,

we need to increase the ticket number by one.

There we go.

Alright, so we have everything. Let's test this.

Let's see if it works.

Looks like there's some errors.

Let's just remove these,

and get them working before we start

'cause otherwise we will run into errors later.

System program/transfer.

This is just us needing to include the system program.

So use anchor_lang system program.

Now let's see if we're missing anything else.

Looks like we have everything now.

Looks like everything is happy.

Yep, everything seems to be happy.

Let us now anchor build it.

So built correctly, now let's go buy our tickets.

I'm gonna go ahead and create another function

for this just so that I can make sure

that I buy the tickets.

I can buy multiple tickets very easily.

So buy ticket. This is gonna be a new function.

We're going to buy all those tickets.

So const, buyTicketIx

equals await program.methods.buy ticket.

And then it's gonna be .accounts token program,

TOKEN PROGRAM ID.

Instruction, now we have to get that blockhash

with context again.

We are going to have to get some information.

We're gonna have to create a new transaction.

So this is gonna be new anchor transaction.

We're gonna do the same thing we did earlier.

There we go. All that code right there.

Add by ticket instruction.

And then we need to just const, signature,

send and confirm our transaction and console log it.

This is going to be the buy ticket signature,

and then we have to actually call it.

So let's go call buy ticket over here.

Await buyTicket.

Alright, so we have our test.

Let us kill our validator,

restart it and buy our ticket.

All right, so it failed and we meant it

to fail actually this time.

So let us see, program fail to complete.

This is on the SendAndConfirm

the buy ticket instruction.

Let's add our Preflight instruction

and so that we can still get the preflight parameters,

so we can still get the signature.

And try this again.

So I'm gonna kill this.

Actually it should be fine.

No, okay, I have to kill it first

and then we should be able to run it.

And what we should see is that there should be an issue.

If we did everything right with the program,

which I might have done a typo here and there,

we should have the signature tell us

that we'll hit a limitation.

So let's go find out.

So if you go over here to our local host Explorer

and see it failed.

We go all the way down to the logs.

We exceeded the CU meters.

So we did, seems like we did everything right

in the program.

We'll figure that in a moment.

But we actually exceeded the amount of CU required

for a base level transaction.

We hit 200,000 compute units.

So this is a great little teaching opportunity right here.

Because we hit the compute unit limitation

on a single transaction, we actually have to increase

the compute limitation.

So let's create two instructions to do that.

So this is something that you will normally do

within your Solana development.

You will normally create different compute instructions,

and priority fee instructions all the time.

So if we do this compute instruction

equals anchor.web3.ComputeBudgetProgram.

Program, not instruction.setComputeUniLimit.

There we go.

We can set a compute unit limit of,

let's make sure I do that right

of, well let's set it as..

So we hit it at 200,000,

let's just set it at 300,000.

We can update this later as needed.

I'd actually recommend, and this is very important, I would recommend updating this units to be as close

as the amount of compute that you need max

for your transaction ever.

This is so that you don't have to worry about this,

and you don't make your users pay more.

All right. So we also need the new priority.

Because we're using the ComputeBudgetProgram,

compute instruction, we also have to create the priority.

So ComputeBudgetProgram.set,

this is gonna be setComputeUnitPrice.

And our price is in microLamports.

We're gonna just set it to one,

normally you're gonna have to set it to something else

or be dynamic with it.

Because we're in a test and only one environment,

we're fine with this.

All right, so now let's add everything

for all of our new instructions that we've just added.

Perfect. Alright, let's run it.

Let's restart our validator and run it.

Alright, so it passed.

Looks like our Buy Ticket signature passed too, so let's go look at that real quick.

I'm gonna go look at...

Click on this and you can see here.

Got a new ticket.

Let's go look at that token account.

It's ticket number zero because we count from zero

'cause we're comp science people.

And it did all the work.

So it set it and verified a collection,

did a clean right to the collection metadata

and added our new ticket to the collection.

So that's all the work that we've done

to create the collection,

add a new ticket to the collection, have someone buy it.

Now the next two things that we're gonna do

is we're gonna do something

with a smart contract called Switchboard.

We're going to do a commit reveal function

to create randomness and use it within our smart contract.

It'd be great.

I'm looking forward to doing this.

Let us get started.

So I'm going to minimize these BuyTicket things

'cause we're going to go into a whole new function.

Let's actually go down here.

Oh no, I can make this faster.

Let's minimize it.

I'm gonna enter here a few times,

and we're gonna create two new instructions.

You can see here we are three in.

These three are gonna be a lot faster, thankfully.

These are all like the setup ones.

All right, so pub function,

we're gonna call this one,

this is gonna be commit randomness.

All right, this is has...

It's try to do some things for me.

I don't care for all this.

Let's just remove it all.

Have a base context one, _context

'cause we're not gonna use it immediately.

And let us create that struct.

So this is going to be a new context,

so I'm gonna create it right here to derive accounts,

pub struct commit randomness.

And then I'm gonna have nothing in there.

And we're going to, let's see,

it's not gonna like it because I had nothing.

We're going to add a few things.

So we know that we're gonna need the token lottery,

and so we need to be able to update that account,

and this is gonna be mutable

'cause we're gonna be updating it.

So mutable, I'm gonna go grab the seeds from it real quick.

So our seeds were right here.

Seeds and bump.

Bump equals token lottery.bump.

And then I'm gonna build this just to make sure

that we are doing everything correctly.

All right, so it built. Perfect.

So there's two things that we're gonna do.

We're going to...

Well, one thing that we're gonna do right here

and then we're gonna do the second thing later.

We're gonna have a randomness account,

we're gonna create that.

This is gonna be an unchecked account

because it's actually gonna be managed by Switchboard,

it's not by us.

So this is managed by the randomness smart contract,

this is specifically Switchboard.

And then we're also going to need the...

Because we're creating an account,

we'll need the system program.

And finally we'll need a payer for this.

So this is gonna be account mutable of pubs,

payer signer info. Perfect.

So those are actually all the accounts that we need.

So let's go back to our function,

commit randomness and get started with it.

So before we do any commit randomness,

we wanna make sure that the clock is done.

We wanna make sure that the lottery is done,

I should say.

The clock is past the lottery time.

Okay. So the first thing that we're gonna do in this commit randomness thing is we're gonna get the clock

because we're gonna need the clock to make sure

that we're setting the randomness information correctly.

The next thing we're gonna do is we're gonna grab

the token lottery account

because we're gonna have to do some checks to make sure

that we're doing everything correctly.

And then the first check that we're gonna have to do,

I believe it was actually the, okay, well, the first check

that we're gonna have to do on ourselves

is we're gonna have to check that the accounts payer,

so accounts.payer.key.

If it's not equal to the authority, we wanna throw an error

that this is not authorized.

Let's go create that error real quick.

So we'll go down here

and we will enter a not authorized. Not authorized.

So we have the error code.

Now this is because we only want the people

that are authorized to commit the randomness,

otherwise someone might do something funky, we don't know.

All right, so context. was, oh

because I have the underscore.

There we go. Fix that.

And the token authority,

this is because it doesn't need to be.

There you go. Alright.

Doesn't need to be referenced.

Okay, so we have the payer checked.

The next thing is we need to grab the randomness data.

And we need to use something from Switchboard to do this.

So let's go add Switchboard to this real quick.

So Switchboard's crate is just cargo add a switchboard

on demand.

So we're grabbing the crate,

we'll be right back in a moment.

Okay, we're here, we have the crate.

Now we can actually use it.

So what we're gonna have to do

is we're gonna have to check this randomness data.

That is what we expect.

So let randomness data equals RandomnessAccountData

and we have to parse it because we need to make sure what's going on.

So accounts.randomnessaccount.data as ref.

We're gonna have to do this...

Actually we're gonna borrow it and unwrap it here.

There we go.

Okay, so we have the randomness data,

let's see what's the problem.

It's because I don't have the type for it.

So we have to go up here,

we're gonna just import it real quick.

So use switchboard on demand,

accounts, RandomnessAccountData.

Okay, let's see.

Saying it's not being used,

it's probably 'cause I've misspelled something then.

RandomnessAccountData,

context.accounts.randomness. There we go.

Looks like everything is happy now.

So we have randomness data and we have to check

that the slot or is a specific...

The slot is valid.

So if randomness data, whoops, data.seed slot

is not equals to clock.slot minus one,

we need to return an error,

ErrorCode, RandomnessAlreadyRevealed.into.

This is because if the randomness data is already,

like we're past the slot basically it's already revealed.

So we make sure that we're not revealed

'cause that would be bad.

Otherwise someone can know exactly who the winner is,

that would be really bad.

So let's go add that, Randomness already revealed.

Randomnesss already revealed.

Okay, so the C is error,

it is just ERR.

Alright, now that we've said all that,

let's go set that in the token lottery.

So token lottery.randomness, account

equals context.accounts.randomness account key.

Okay so we have the commit randomness,

we have everything here that is required for it.

Now there's a few things that we're gonna need,

and it's quite a lot of things actually.

Now that we're using Switchboard,

we have a new Switchboard program that we're building,

or that we're using.

We're gonna have to make sure that we add all those

just like we add the specific metadata program right here.

And there's a lot of different accounts that you need.

I've made it easy for us to develop with it,

if you go under to project 8 token lottery setup,

there's gonna be two separate SH files.

All you have to do is go into it and copy and paste it,

and create your own SH file.

So I'm gonna do this right now.

We're going to create a bunch of things here.

So I'm gonna go make a new directory.

This one's gonna be called setup,

and go into setup directory.

This is just gonna be just like how this is set up.

There's a setup directory,

and we're going to create in that directory,

I'll stop doing the stuff all in the CLI for just a moment.

We're gonna go into that setup directory and create two files.

So this is gonna be setup-local.sh

and then I'm gonna copy and paste that

from the setup local file.

And the other one is going to be the set up start validator.

This is just gonna have that really long command

that we need to set up all the accounts.

Alright, so and the next one says start validator.

Alright, so we have both of these shell scripts now.

You can look at 'em if you want.

I would recommend always looking at and checking it.

I'm gonna make them execute all of chmod set up local

and chmod plus X start validator.

And then we're gonna run them.

So set up my local,

this is just gonna download all those accounts

that we need from mainnet.

Now note, you should only have to do this once.

If you run into some errors

that were unexpected throughout this,

you might need to do this again.

It's all because like the randomness accounts might expire and that should be fine.

All right, so we have all the different randomness accounts.

Now we can just run start validator.

We'll just run this again over and over.

This just will grab all the accounts that we need,

and start it in our local validator.

Alright, so next thing we're gonna do

is we're going to actually run our script.

So I'm gonna buy a couple tickets

just because I wanna make sure

that I have a lot of tickets later

so I can test some things.

Alright, so we're gonna buy a bunch of tickets

and now we can do all the different things

for running this commit randomness function.

So the first thing that we're gonna need

is we're gonna need the queue.

So the way that Switchboard works

is there's kind of like a queue of randomness accounts

and it goes through this queue

and it gives you this randomness data.

So I'm gonna set up this queue,

new anchor.web3.public key.

This is going to be a specific value.

It's gonna start with...

If you go to, I'll show you

how you actually figure all this out.

So go to Switchboard on demand, their code.

So this is their randomness code right here.

This link will be in the GitHub repository.

If you go to their specific,

I believe it's go to their index,

they're gonna have a whole bunch of code here.

You'll see it. It's very similar to what we're about to do.

And if you go to their utils, they'll give the specific,

let me make this bigger for y'all,

they'll give the specific queue thing.

We're gonna work with the queue from mainnet

'cause we copied it from mainnet.

So let's go ahead and copy and paste that account.

Alright, so this is gonna be the queue.

The other thing that we're gonna need

before we start doing anything else

is we need to actually pull the specific code,

or the program to run with a Switchboard program

'cause we're gonna do a bunch of things

to set this up in the JavaScript side.

So let's go ahead and do that.

So we have this Switchboard program.

I'm gonna do a let switchboardProgram for now

and we're gonna update that in a moment.

We're gonna need this later,

so I'm gonna go ahead and create it.

This is gonna be a randomness key pair.

So anchor.Keypair.generate

and then we're going to create the specific...

We're gonna grab initial or grab switchboard.

We're gonna do a before, and this before is going to load

the switchboard program, Load switchboard program.

That's gonna be an async. Perfect.

Alright, so we have the before,

this what it's gonna do is we're gonna do const.

We gotta grab that switchboardIDL

equals await anchor.program,

capitalize Program, fetchIdl and we have to fetch it

from the switchboard specific program.

So we have to include a Switchboard from here.

So let's do a npm.

Is this the right place?

It should be in the thing.

So npm add or install

at switchboard/ or switchboard-xyz/on-demand.

This is Switchboards nice package for this.

At switchboard on demand. Perfect.

So the ID is gonna be switchboard.switchboard on demand.

So the switchboard ON DEMAND PID,

that's the program id.

And then we need to set the connection.

So this is gonna be from,

this is actually gonna be from mainnet.

So connection, new anchor.web3.connection.

And then the provider or the URL we're gonna use

is gonna be mainnet.

So if we set this real quick,

I'm going to make sure I remember.

We're gonna set this to the mainnet URL.

So I'm gonna go grab the mainnet URL real quick

from the Solana Docs.com/docs.

And this is gonna be the RPC URL, so let's go grab that.

So mainnet RPC is gonna be under that public endpoints.

So let's go out, here it is right here.

Put that right there, and now we have the IDL.

So then we can do switchboard equals new program

using this switchboard IDL.

And let's see, make sure I did everything correctly.

Program, new switchboard that should be the provider,

not the PID, provider 'cause we're grabbing it

from our provider.

All right, so this error right here

is just because it's saying like, hey this might not exist.

We wanna make sure it always exists and that's okay.

So I'm gonna go ahead and put that as anchor IDL

and we should be fine.

So as anchor.IDL.

Right, everything's fine.

This should load that program so that we can use it later.

Now what's interesting

is we can actually save this to a JSON file

and we're gonna do that in just a bit,

so that we don't have to constantly load this over and over.

So I'm actually gonna go grab this ID

and we're going to pull it out and set it as a JSON.

So be right back.

Alright, so I did some quick coding.

All I did was I used FS to save that switchboard.json

to my file system.

That way I don't have to do this over and over again.

Now what I can do is I can just grab that IDL

and just be fine.

So the switchboard program,

we're gonna just make it that a const instead.

I'm going to go ahead and comment out this code.

What I'd recommend, I'm not gonna put this JSON file in here

because it might change but always try

to get it again later.

So we're gonna do new.anchor.program.

This actually says that the switchboard

on demand IDL exists.

It doesn't look like it does, so that's okay.

We're going to have to import this real quick.

So just like we did the target types token lottery,

we're gonna have to do import switchboard IDL.

So switchboard IDL.

From this is gonna be from that file that we just did.

So I saved it to switchboard.json

and let's make sure that it updated it correctly.

So for Idl's which are saying not found,

this is because we are in tests so it needs to go backwards.

So backwards and there you go, I found it.

All right. So we have switchboard.json,

this one's gonna be as anchor.Idl

and then it's gonna be the provider.

And let us make sure we did everything correct now.

I believe I did everything correct.

We should be able to just anchor test it

and well since I've run this test before

and I haven't restarted the validator,

we expect the test to fail but I don't want 'em to fail

on the switchboard program.

Let's see what happens.

So it did fail, perfect,

and it failed on commit confirmed transaction.

So now we have the switchboard program. Perfect.

All right, so I'm gonna remove all this extra code.

Whoops, didn't remove all of it.

There we go from top to bottom.

And now we can start testing our commit reveal function.

So we've reloaded the switchboard program,

we have to go all the way down here with our queue.

We have to start doing things with this specific queue.

All right, so we need to go grab our queue account now.

So que account equals,

this is gonna be new switchboard.que

and this is gonna use that switchboard program

that we used earlier and the queue that we just pulled.

All right, so the next thing we need to do

is we need to get the queue accounts.

So we're gonna do const, queAccount

equals new sb.que.

This is gonna be with the switchboardProgram

and that specific queue that we have right above.

There we go. And we're gonna do a quick try catch.

Here we go, let's just console

and we'll do a process exit,

so that we can see what the error was.

We don't wanna continue just in case something went wrong.

And then here what we'll do

is we will do await, queAccount.

Let's load data, that way basically we're loading

the data into the queue.

Alright, the next one we're going to do,

we're gonna do our create randomness

and do our commit reveal.

So const, randomness, createRandomnessIx

equals await sb.random,

whoops, Randomness.create.

This is with the switchboardProgram,

the rngKP, and our queue.

Okay, it looks like...

Oh I forgot a equals. It's fine. There we go.

And then we gotta create our instructions.

So createRandomness, well first in transaction.

So await sb.asV0 transaction.

Connection, we gotta do the provider.connection

'cause our current provider,

this is going to be our ixs

which are just going to be our createRandomnessIx

is instructions.

Payer is our current wallet, right?

So wallet, not payer, but publicKey.

There we go.

And finally the signers are gonna be that wallet.payer

plus the rng keyPair that we created.

Alright, so the next thing that we'll have to do

is we have to send the transaction.

So const, createRandomnessSignature

equals await provider.connection.send transaction

with our specific createRandomnessTransaction.

There we go.

And let's just console.log,

createRandomnessSignature,

so that we can see what the signature is.

All right, so let us restart our validator

and do our anchor test.

Oops. So anchor test, skip local validator.

There we go. And see what happens.

All right, so it passed.

Let's go grab that signature.

So here's the create randomness signature.

We'll go back to here,

we're gonna open Explorer as usual,

go make sure it's on local, go to that signature

and we can see here we had our transaction it was using

so we did the compute limit, that's our program.

Create, create, we got the account size

for the immutable owner or sorry for the token program.

We did a create account.

Initialized, initialized.

So here's a address lookup table thing.

This is for something that we did

in the Switchboard program.

So it's allocating, it's assigning.

So this will allocate a sign and transfer.

This looks like this is about what we did

for our queue account.

So there we go.

All right, so we have our...

That one worked.

All right, so the next thing that we need to do

is we need to actually do that commit,

so that we can commit the randomness to be later revealed.

So let's go ahead and do that.

All right, so we do const commit,

we're gonna create that commit instruction.

It's gonna be await program.methods.commitRandomness,

this is that commitRandomness instruction

that we created earlier.

We're gonna have only that account that we just created.

So that randomnessAccount,

randomness.pubkey. There we go.

And we're gonna make sure that we pull this instruction out.

All right, and then like before,

we're gonna do const.computeIx.

This is gonna...

We're gonna call it a commitComputeIx

equals anchor.web3.ComputeProgram.

There we go. And we're gonna have to set the units,

let's set it to 100 K.

I don't think it's gonna need 300 K,

then we're gonna need that priority fee prior.

So this is going to be the commitPriorityIx set Compute.

This is not compute enterprise.

Oh no, it is compute enterprise of micro lamports.

We'll set it as one.

We're working on local, that's okay.

And then we're going to get that blockhash.

So commitBlockhashWithContext,

await, getLatestBlockhash

and then we have to create that instruction.

So const, commit or the transaction.

So new anchor.web3 transaction,

it just filled in everything.

Let's see if it filled in.

Fee payer, provider wallet,

public key with the commit the blockhash

and the latest lastValidBlockheight is that.

And it just added our IX.

We actually need a little bit more.

So we need to add this IX. Whoops.

We need to add our new...

The commitComputepriority, ComputingIx.

And then finally the commitPriorityIx.

And there we go.

So we have commit, the Compute budget priority, commitIx.

We also need the sb commit IX.

So add sb.

Oh, so we didn't actually create the Switchboard command.

So we have to also create that.

So commit IX, this is...

Because we created our randomness account

without create randomness signature,

we need to also make sure the Switchboard program

is using that randomness account to commit for it.

We need to provide this in the same instruction

that we do the commit on our actual program.

So this is gonna be Switchboard commit IX

equals await our specific randomness account.

Commit IX with our queue. Perfect.

So this needs to go before our new commit IX.

So I'm gonna put it right here, right beforehand.

Whoops, not that one.

It's sbCommitIx.

There we go.

And don't have a semicolon. There we go.

So we have our four instructions that we need,

and now we just need to send and have a signature.

So commit signature.

What we're gonna do is send a confirmed transaction.

It's gonna be provider, our commit transaction.

We're gonna have our...

So commit transaction wallet.payer.

I'm going to have skipPreflight on

or sorry, skip Preflight not on

because I just wanna get the signature

and then we just need to console log it.

CommitSignature. There we go.

I just realized I have to restart the local validator.

So always remember to do that.

So restart it and then run it.

All right, failed.

So let's figure out why I failed.

Should test, cannot read properties of undefined of key.

This is for queue.

Looks like I missed something on the queue.

Let's go look that up real quick.

All right, so I figured out what the issue was.

So what happened is some of my accounts, they expired.

If you actually dig into the code for a Switchboard,

they expire after was like six hours.

So mine have expired.

So what you kinda do is just do set up local,

you run this and then you restart your validator.

So let's just do that real quick. There we go.

Start a validator and then that should run it.

And then let us anchor test skip local validator.

And now as long as it doesn't like exceed

the test timeline, that should pass.

Alright, let's see what the failure was.

Account does not exist or has no data.

So try to pull some account.

Let's just make sure that, yep,

so I'm not actually setting it.

Let's run this again just to make sure.

All right, so the issue here

that we ran into where it's failing

is actually because this send transaction didn't wait

for it to confirm until we moved on.

And since this is using a special type of transaction,

it's called a Versioned Transaction.

We're gonna do a little bit of extra work

in order to check it.

So we're gonna do this real quick.

We're gonna do a let,

let's call this confirmed equals false

because it's not confirmed yet.

And while it's gonna be a simple code

that we probably write in our college days,

I don't want it to throw, let us take that out.

We basically just need to check if it's confirmed.

So const, confirmedRandomness

equals await provider.connection.getSignatureStatuses

with our create randomness signature.

And then so randomness status,

we need to pull that specific status out.

This is not quite right.

Let's just remove that extra code right there

says .value 'cause it will be zero

'cause it's the only one that we send back.

If randomness confirmations is not equal to null,

so not undefined.

And the randomness confirmation,

let's do randomness confirmation status

is equals, equals, equals to confirmed,

then we set confirmed equal to true.

So let's clean up some of this code.

There we go.

And clean that up a little bit as well.

All right, so if confirmed equals true, then move forward.

If not, continue looping. There we go.

All right, so let's kill this validator,

restart it, start it here.

Whoops, there we go.

And so this should always work now

because what it's doing is it's now waiting

for that transaction to be confirmed before it does

the overall commit transaction that we just coded up.

There we go.

And sure enough, it got confirmed.

So one thing that you also note

is that our time has increased dramatically.

What you do is you just add a timeout.

I've made it a very large timeout.

The real thing that you would want to do

is like separate these tests,

but because we're just doing one large integration test,

it's all right.

So now we've done that, we've done the commit.

Now we actually have to code up the reveal.

So let's go back over to our smart contract code.

Let's start writing up the reveal instruction.

Alright, so we're gonna create first the derive accounts.

So the accounts that you would need

for this specific reveal instruction.

So this is gonna be,

we're gonna call this RevealWinner info.

There we go.

So what are the things that we need for revealing?

There's a number of different accounts

that we're gonna have to consider.

Things like the commit account.

We're obviously gonna need the payer to do things,

but overall what we need is,

so let's start writing that actually up.

So first we need that payer

because there's gonna be some money being used.

The next account that we're gonna need

is, oh, figured it already.

We're gonna need the token memo or sorry,

the token lottery.

So there we go. There's our token lottery.

And then finally we need the randomness account

that we're using.

So this is gonna be an unchecked account

because it's going to be created

via the Switchboard smart contract.

That's okay. Every time you use an unchecked account,

you should always have some red alerts going on

and make sure that you know what you're doing

when you're using this.

I know that this account is being created and checked by the Switchboard smart contract, so I'm okay.

Alright, so we have the reveal winner accounts,

now let's actually create the instruction.

So we'll do pub function reveal winner,

we have to create that context with the context

that we just wrote.

So this is gonna be with that RevealWinner context.

And then we don't need anything else.

So we're gonna do outputs the same thing as usual.

We do outputs of result.

There we go. And we know that it's going to give an okay

at the very end.

Oh, all right. I'm gonna just go ahead

and do an anchor build while I code the rest of it,

just so that if I run into any issues,

this will tell me very quickly.

Let's see. All right, looks like we didn't run any issues,

so let's start writing it.

So first thing, we're gonna have to check the clock.

So get the clock.

Then we're going to have to do, let token lottery and get our specific token lottery account.

We need to make sure that the token lottery account,

so accounts.payer.key, that the person who is doing this

or revealing it is the specific authority

of the token lottery

'cause otherwise we wanna return an error code

of not authorized.

We don't want someone revealing that they're not allowed to.

And the next thing that we need to do

is we need to make sure that the specific account,

oh, is that right?

What am I doing right there?

Contexts.accounts, let's see, something is missing there.

Similar name for context. Let's see.

Contexts.accounts.payer, should be right .accounts.

All right, sure.

Context.accounts. We need to make sure

that the randomness account is the same randomness account

that we set earlier in that token lottery config,

otherwise we need to exit because that is a problem.

It's avoiding someone setting a new randomness account

and hijacking it based off of a randoms account that creates a good random value that will makes them win.

We wanna make sure

that it's set before everybody buys tickets.

The next thing that we need to do

is we need to check that the end time is set,

otherwise the lottery is not completed.

And then the last thing that we need to do

is we need to also require,

this is another way that you can do these checks.

I think it's kind of cleaner.

We need to require that the winner is not chosen

because if the winner's already chosen

then what are we doing?

Why are we trying to overwrite with a new winner,

otherwise we can accidentally create more winners overall.

Alright, so randomness data.

When you grab that randomness data

from the randomnessAccountData,

so we're gonna have to parse it.

This is grabbing from the context account's,

randoms account data borrower.

This is so that we can actually get the information

out of that account and that we know what to do.

So we're gonna get that revealed random value.

Oops, there we go.

And it's coming from that randomnessdata.get_value

and not the clock.

Well, it is the clock but it's a reference to the clock.

Otherwise you can't make good Rust code.

Then we're gonna have to do is we're gonna have to make sure

that we candle an error.

If there is an error, you need to make sure, oops,

that our randomness is not resolved.

So say that someone's trying to reveal this winner

in the same instruction that, or same transaction

that they're committing, we should fail

because it must be that clock plus one.

It must not equal basically.

So next thing we need to do

is we have the revealed random value,

we need to create the winner.

So the winner equals revealed random value

of the first random value in that array.

And we're gonna do some...

We're gonna make it as U64

and we're gonna do some basic math

that we are used to in computer science.

We're gonna do it over the total tickets.

So we don't want it to reveal value to be larger

than the total ticket amount

'cause then we just run into issues.

So now we have the winner,

and we need to make sure that we set the winner

of our token lottery and we need to say

the winner was chosen equal to true.

And that is it. That's everything that we need

for the specific token lottery revealing the winner.

Alright, so let's do an anchor build and see if it runs.

Alright, so we've done that.

Now let's go over to our test

and start writing that test again.

So after the commit,

we want to now use our new reveal winner instructions.

So revealWinnerIx equals await.

Now this is not gonna be all the code thanks to Copilot

for trying but not quite.

It's not the programId,

but we're gonna need one account set here

and that is that randomness account.

Otherwise it won't know what random's account to use.

The next thing that we're gonna have to do

is we're going to do what we did before.

We're gonna have to create a revealBlockhashWithContext

and then const revealTx equals new anchor transaction.

Then we have to set the actual variables just as before.

There we go. This is that fee payer,

the provider using the specific things

in our reveaVBlockhashWithContexts.

There we go.

And we have to add two things.

We have to add the revealWinnerIx.

And then there's one other thing

that we haven't done yet

is we actually have to call Switchboard

and make it reveal as well.

So right before we do all this,

we're also gonna just do a const sbRevealIx

and we're gonna reveal that specific randomness

off that account.

So we have to do that first,

otherwise it doesn't know what to do.

So let's add that.

So this one is going to be add revealIx

and then add the specific reveal winner.

Let's just clean up this code just a little bit,

so it's a little bit more readable.

There we go.

So now we have the reveal transaction,

and we need to just send it.

And one thing to note is you cannot send

the reveal transaction so close to the slot

that it was committed in.

So we need to make sure

that we're not going to reveal it too soon.

So we're gonna set the current slot

and then just like before,

we're gonna do while currentSlot,

sorry, we don't want the slot to be before the end slot.

So there we go.

We're gonna just do a const slot

equals await get current slot.

If slot is greater than currentSlot,

then currentSlot equals slot. Whoops.

There's a lot of extra code there.

And then we just need to finally do console.log.

I'm gonna just log the current slot

just because I want to know what's going on.

All right, so what this is doing

is, is waiting until the end slot

and it'll just continue on until the end slot is hit.

And then once it's hit we'll be able to do this reveal

'cause we can only do the reveal after the lottery is ended.

All right, so do const revealSignature

equals anchor, sorry, await, anchor.web3,

sendAndConfirmProviderConnection,,

reveal transaction, wallet provider.

And then let's console log it. Perfect.

All right, so we have the console log.

Now let's see if it runs.

So I'm gonna go ahead and kill this

and let's also check,

make sure that if my memory serves correctly,

we wanna make sure that our configuration

that we set early on is not gonna be too crazy.

So this initializedConfig needs to start end.

It looks like that we did the current slot plus 20.

So we should be okay.

I just wanted to make sure I didn't set the slot too high,

otherwise we'd be waiting forever.

All right, so kill the local validator,

rerun it and then we're gonna run this test.

So let's go ahead and run it.

Whoops, that's anchor build.

Might as well let it build.

Alright, let's run the test and see what happens.

So you can see it did some work on the slots,

and then it eventually did a reveal.

So let's go look at what that reveal looks like.

Alright, so if our memory is correct,

we can see here this is the Switchboard program and it did a create an account.

And this is our program.

So we can see that it did the randomness reveal

on that Switchboard program and then it revealed

the winner on ours.

And there you go. So we have the winner.

Now what we need to do is we need to make sure

that we claim the money out of that account.

So we have a winner

and let's actually, let's just add some...

A little bit of code to see who that winner is,

otherwise we won't be able to see who actually won.

So I'm going to go ahead and hint here.

We're gonna add a log real quick.

So message, we're gonna set that winner

and I wanna see who that winner is.

So let's go and kill local validator again, restart it,

and then run our test and go look at that in signature.

All right, so let's grab that signature

that we just created, paste it, just make sure...

I think, yeah.

Okay, let's paste it in. Boom.

And let's go look at the logs.

So our winners chosen is three.

So the third buy ticket out of all the buy tickets

we did up here, so we did a whole bunch of buy tickets

right here.

The third one if we...

I think we're counting at zero.

So the fourth one will be the winner.

So there we go.

We have a winner,

and it's within our range of all our tickets.

All right, so now we have the winner. It is set.

Let's actually write the code to do the claim now.

So I'm gonna do some cleanup real quick

where we revealed, I already revealed.

We have this token lottery, perfect.

Bunch of error codes, okay.

So we're gonna do the normal thing of,

we're going to derive,

we're gonna derive the counts, pub, struct,

and we're gonna call this ClaimWinnings

with the right caps.

Winnings. Perfect.

And so we know that we're going to need, as usual,

we'll need the payer.

So pub, payer signer info.

The next account that we're gonna need,

we're gonna need quite a few accounts

because what we're gonna have to make sure that we do

is we have to grab the ticket account,

we have to check that the ticket belongs

to the specific collection that is set within our config.

And then we have to actually release the funds.

So we have to also account for that account as well.

Lots of accounts, so let's start listing them.

So the first one that we're gonna have to do, so count.

First one is going to need our token lottery

'cause that's an easy one.

We don't have to really think about it.

There we go.

So this is pub, token lottery, Account, TokenLottery.

The next one that we're gonna have to do

is we're gonna have to grab the ticket mint.

This is a specific ticket that the user has.

So let's do mutable.

Actually it's not mutable

because we don't need to change it.

But the seeds are, this is to...

We have to make sure that it equals

the token lottery winner seeds.

So this is the third one that we saw earlier.

It's gonna be different every time,

but in the previous example it was third.

So pub ticket mint,

this is an InterfaceAccount, info mint.

Because we have that mint,

we know that we're also going to have the specific...

We're gonna have a token program.

So let's go and put that there.

So, pub token program.

This is gonna be interface of TokenInterface.

All right, so we have the ticket mint.

The ticket mint belongs to a collection

that we're gonna have to check.

So let's do that one as well.

There's lots of accounts.

So this is gonna be the collection mint.

So seeds equals, this is gonna be that collection mint seed.

So b collection mint as ref bump.

And this is pub, collection mint,

InterfaceAccount mint.

The next one that we're gonna need

is we're gonna need some accounts

from the token metadata program.

So before I do all those, I'm gonna go ahead

and put down here the token metadata program.

So, pub token metadata program,

program info Metadata.

Metadata is the type for the program, right?

The struct for the program.

Alright, so let's go and grab those accounts now.

So whoops, account.

We're gonna first grab the ticket metadata.

So the ticket metadata count.

So seeds equals, this is gonna be b metadata,

token metadata program as ref.

And then the token, the ticket mint key.

So it's specifically the metadata on the ticket mint.

So then the seeds come from the program

of the token metadata program.

There we go.

And this is going to be pub ticket metadata.

This is not in an unchecked account

because we know it's already created,

so we don't have to do anything special with it.

So this is an account of type info,

well type MetadataAccount.

There we go.

And then the next one that we have to do

is we have to check the winning ticket account.

So we need to go check

that associated token account real quick.

So let's go grab that.

So account, this is gonna be seeds equals,

let's see, oh sorry, not seeds.

We can just use the special things

from associated, token mint equals the ticket mint.

Associated token authority equals the payer.

So it's owned by the person who is sending this instruction.

And then the token program

is of the token program that we expect.

This is the ticket account that we expect.

We're not just using an account,

we're gonna use an InterfaceAccount.

There we go. And then finally we need

the collection metadata.

And the reason being is we need to make sure

that this ticket mint belongs to the collection metadata

that we set.

So account. And then these are seeds.

Let's see, yep, it knows exactly what I'm gonna do.

The metadata, token metadata program

and the collection mint,

the seeds are from the token metadata program

and pub, winning, sorry, collection metadata

equals the specific Account info MetadataAccount.

All right, so we have everything here,

now we actually have to write the claim instruction.

Alright, so let's go ahead and write it.

So we're gonna do pub function claim winnings of context,

context of our ClaimWinnings context.

We're gonna do the same thing as we usually do,

outputs a result.

And we know the end of this code is always the same.

There we go.

Let's make sure I spell claim right,

otherwise we're gonna run into issues later.

Okay. So first thing we need to do is we're gonna require

that the context.accounts.token lottery winner chosen.

So if the winner is not chosen, we want to exit out.

So we wanna make sure

that the winner's chosen before you claim the winnings,

otherwise you can do something funky.

The next thing we're gonna require

is two things on the metadata.

So the collection metadata stuff.

So first we're gonna do

the accounts.ticket metadata collection.

We're gonna make sure that the collection is verified

'cause if it's not verified, that means you can't guarantee that the ticket is part of the specific collection.

So we have to make sure that the collection is verified.

There we go.

If it's not verified, we want to output

the error not verified.

The next one, we wanna make sure

that it's actually part of that specific collection.

So .ticket_metadata.collection as ref.

So let's make sure all this is right, this is not right.

So let's just get rid of some of this code. Boom.

So as ref unwrap.key.

So we have to make sure that the key equals

the context accounts.collection mint.key.

If not, it's the incorrect ticket

so it doesn't belong to this collection.

All right, now we have to...

For some work on the front end side,

we have to make sure that the ticket NAME.to owned.

Let's actually check what is this name?

All right, so this name is that ticket name.

So we have to make that this ticket name,

we have to make sure it matches the name that is expected.

So that number three needs to be ticket number three, right?

So we have to do some work on concatenation.

So we're gonna just do ticket name

to owned context.accounts.token lottery winner to string.

So we have to add 'em together. Whoops.

Token lottery winner to string. There we go.

And why is it failing?

Let's see, our unexpected semicolon.

Oh I have a semicolon set equals, there we go. And save it.

There we go. Go do a couple enters.

Alright, so let metadata name equals,

this is gonna be the context.accounts.ticket metadata.

We're gonna have to do name...

We're gonna have to remove some of the extra valuable,

or sorry, extra values.

Otherwise we're gonna get some weird values in our name.

You can go ahead and try this without this code

that I'm writing now.

And you'll see that it'll give a lot of like extra values

that didn't belong.

Right. There we go.

So we have the ticket name, the metadata name,

and now we need to require that the names match.

So require, this is gonna be the metadata name

equals the ticket name.

If not, it's the incorrect ticket.

And we also have to make sure

that the specific winning ticket account is actually there.

So they didn't just create a token account,

and doesn't have any tokens in it.

That would be bad, right?

Because they didn't buy the ticket,

they just created an account

to hold the ticket without the ticket.

So let's make sure that exists.

This is ticket account.amount,

it's greater than zero.

And if it is greater than zero,

if it's not greater than zero, we need to say no ticket

because they're trying to cheat.

So next thing we're gonna have to do,

we're gonna have to just basically transfer

the lottery pot amount.

So accounts.token lottery.to account info.

We have to pull out those lamports from our account.

Borrow, let's see,

minus equals the contact accounts.token lottery,

lottery pot amount. There we go.

And then we have to transfer it out to the winner.

So payer to account info plus equals the lottery pot amount.

And then finally we need to set

the lottery pot amount from token lottery

so that they don't just keep draining our lottery

amount or something.

Say we start a new lottery with the same amount

or something, we wanna make sure

that they're not just constantly draining.

There we go. All right, we set to zero.

Let's do an anchor build, see what happens.

All right, awesome, it built.

Now we have to actually write the test

for this claim, right?

It says write this test real quick.

So we're gonna do const claimIx

equals the await program.methods.claim winnings.

This is gonna used with that specific account.

So accounts, we're gonna have to make sure

that we use the specific token program.

So token program equals,

we're gonna just use the normal token program,

not the extension token program.

There we go. They have instruction.

Now we have to do the same thing we've been doing.

So const, claimBlockhashWithContext and then const,

claim transaction equals new anchor.web3.Transaction.

There we go. Create all of our fee payer stuff.

Add the specific claim instruction.

Let's see, it added an additional thing there.

So we have our instruction now.

Now we pulled the signature.

There we go. And finally we log it.

Alright, let us restart our validator and run this test

and see if it works correctly.

So we restarted the validator,

now we just need to run the test.

All right, so it failed.

Let's figure out why it fails.

Failed, error BC4.

Okay, so what is BC4?

So we gotta reveal, transaction simulation failed.

This is on the sendAndConfirm with the claim.

So let's go see what BC4 is.

Hex to decimal.

This is one what you do all the time

when you run into these weird values.

3012. So anchor error 3012, let's see what it is.

Account not initialized.

So it's saying some account that we have

is not initialized yet.

Let's go figure out which account it was.

Let's look at this real quick.

Let's see. The winner chosen was, is that three?

Let's see. Oh, it didn't tell me.

So let us go ahead and skipPreflight.

There we go. And run it again.

So we'll kill these, and then we have to skipPreflight

and make sure that it's all good.

Okay, so let's go grab that signature.

We grab that and go put it in the Explorer.

The reason why we do this

is we get a little bit more information.

We can see here, oh, the ticket metadata,

AccountNotInitialized.

So this program are expected.

So it says that our ticket metadata was not initialized,

which is very interesting.

We have the token, but the ticket metadata,

the ticket metadata was not initialized.

So let's go pull that and double check

that I wrote all the code, right?

So we go over here.

We go. Ah, there we go.

It's just metadata.

We miss milled metadata.

That's an okay error. I'm okay with that one.

So let's go ahead and run it again.

And this time let's see if it completes.

All right, so passed.

Let's go look at that claim,

make sure everything worked as expected.

So there we go.

There's our claim signature.

We can check real quick.

Oh, there we go. 0.65.

So we gained some SOL over from the specific claim...

The claimed winnings.

It went 0.65. It should be the number.

Let's see how many of our tests did we buy tickets?

How many tickets did we buy?

1, 2, 3, 4, 5, 6, 7. Is it seven?

So yes, this is seven.

So seven times. Was it seven times 5,000?

Is 5,000 the amount that we said?

No, it was 10,000, right?

Let's see, where is our specific initialized instruction?

Oh, so there's our buy,

there we go was 10,000.

So seven times 10,000, minus 5,000

for the lamports for the fee

'cause this is our fee.

So we can see that the seven,

there we go was transferred from our token lottery account

to our account. And we good.

We're good. We did the claim.

So congratulations.

We have created a token lottery.

And just to generally recap what we just did,

what we did was we created a token lottery

where you bought tickets that were tokens in a collection

in order to enter the lottery.

Those tokens created a fee

that went to a lottery pot account.

That lottery pot account was then used later

to claim the winnings when we claim the winner.

To figure out who the winner was,

we used Switchboard random dysfunction.

So we used some on chain randomness,

is not truly on chain, it's off chain,

but we're using a commit reveal function

so that it's easily verifiable.

And we use that to create our randomness, choose our winner,

and then create it in a trustless manner and give the winnings to that winner.

So that is a lot of different things that concepts

that we've learned throughout this token lottery project.

We will be using this in the future in the Bootcamp.

So look forward to that.

If you have any questions or run into any issues,

just check the GitHub repository,

join the discussions there.

But thank you and let's go to the next one.