Solana Blockchain Developer Bootcamp Day 2 Exercises

Exercise 1: GM Anchor Project

In this exercise we will create, deploy and interact with a solana program that takes your name as an input, and simply outputs a message to the program output. This exercise is exactly the same as the first exercise from day 1, except this time we will use Anchor. This exercise will teach you about the basics of using Anchor in your Solana smart contracts.

Installing Anchor and Initiating the project

1. First step is to check if you have Anchor installed. Run the following command and see if you get back some meaningful output or not:

If you get back the anchor version number, then you have Anchor installed, and you can skip to the next step'. If the command isn't recognized, you need to install anchor:

First you need to install Yarn if you don't have it installed

```
yarn -v
```

If you don't get a version back, you can install yarn via the following command:

```
npm install -g yarn
```

Now you're ready to install Anchor

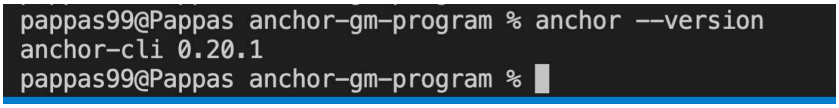
```
cargo install --git
https://github.com/project-serum/anchor --
tag v0.20.1 anchor-cli --locked
```

On Linux systems, you may need to install additional dependencies if cargo install fails:

```
sudo apt-get update && sudo apt-get
upgrade && sudo apt-get install -y pkg-
config build-essential libudev-dev
```

Now check to ensure Anchor is successfully installed

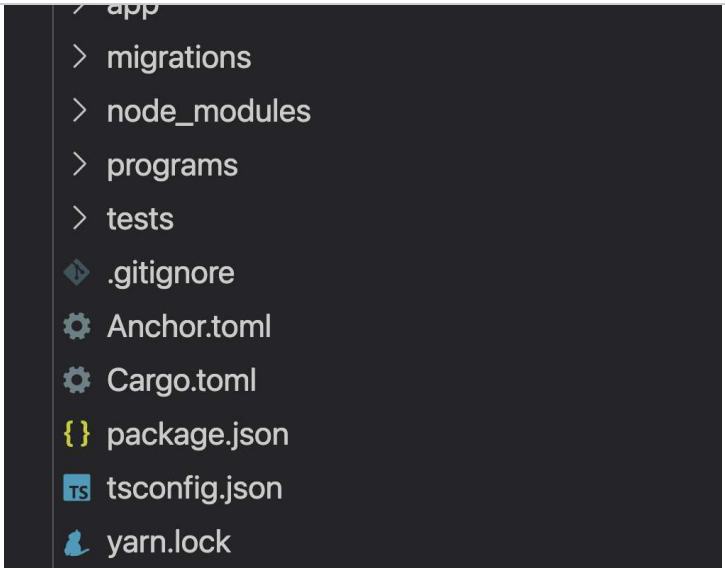
```
anchor --version
```



2. Now that we have Anchor installed, the next step is to initiate a new project using the Anchor CLI. Ensuring you are in the folder that you want your exercise folder created, enter the following command into the terminal. It will create the necessary folders and files for your Anchor project. Once created, ensure you can see the folder structure in your VS Code folder explorer. In this example we enter 'cd ..' first to go back a folder, as we were still in the folder of our previous exercise:

```
cd ..
anchor init gm-anchor
cd gm-anchor
```

You should see a 'gm-anchor' folder created that contains all the necessary files and folders for your Anchor project. Ensure you can navigate this folder structure in the VS Code



The next step is to install the minimist npm package. We will use it when we create the client later on

```
npm install minimist
npm install
```

3. Now we need to tell the Solana CLI that we want to use a local cluster. Note, If you're on Windows, it is recommended to use WSL to run these commands

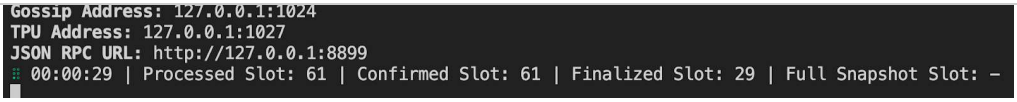
```
solana config set --url localhost
```

4. Next step is to start our local cluster. Note, you may need to do some system config and possibly restart your machine to get the local cluster working. Type the following command into the terminal. Note, you may want to open up a second terminal for this command, so you leave your existing one available for future CLI commands:

```
solana-test-validator
```

If you're using WSL and a build from source, you need to run the local validator not via the Solana CLI:

```
cd ../solana/validator
./solana-test-validator
```



If you can't get the local validator running, then you can just use the public Devnet network. You can configure your setup as follows. Please do not run this command if you are able to successfully start the local validator.

```
solana config set --url
https://api.devnet.solana.com
```

5. Now we can create a new CLI Keypair. We will use this for interacting with our local cluster.

```
solana-keygen new -o id.json
```

6. Now that we have a new keypair, lets use the Solana airdrop program to retrieve some lamports that we can use to pay for transaction fees:

```
solana airdrop 2 $(solana-keygen pubkey
./id.json)
```

7. Now that we have a new keypair, let's also explicitly tell Anchor to use this keypair when creating transactions. We will do this by setting the ANCHOR_WALLET environment variable. If you are using windows, you may need to set these in your system environment variables. Note we're appending a '..' to it, because our client will be in a folder under the project root folder:

```
export ANCHOR_WALLET='../id.json'
```

You're now ready to start building the on-chain program!

Creating the Program

In this section, we'll create a new Rust program that takes in a name parameter, and says 'GM' to that name by outputting text to the program output, and storing the name in an account which is then read by an off-chain client.

8. The first step is to open the programs/gm-anchor/src/lib.rs file and enter in the following code. This code
    a. Defines a new anchor program

the accounts passed into the execute instruction, and the deserialization of the gm_account account into a 'GreetingAccount' struct

d. Defines the GreetingAccount struct, that stores the name string

Note: If you are using the latest version of Anchor (0.23) ,replace the text **ProgramResult** in the execute function definition with the text **Result<()>**, as the syntax has changed. You can check which version of Anchor you're running with the terminal command: *anchor --version*

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

declare_id!
("75PsQyHnFLhmF1jb4m31f2Gt35HFou3vJsaBCVAhYKAo");

#[program]
pub mod gm_anchor {
   use super::*;
   pub fn execute(ctx: Context<Execute>, name:
String) -> ProgramResult {
       let gm_account =
&mut ctx.accounts.gm_account;

       gm_account.name = name;
       msg!("GM {}", gm_account.name);
       Ok(())
   }
}

#[derive(Accounts)]
pub struct Execute<'info> {
   #[account(init, payer = user, space = 8 + 32)]
   pub gm_account: Account<'info,
GreetingAccount>,
   #[account(mut)]
   pub user: Signer<'info>,
   pub system_program: Program<'info, System>,
}

#[account]
pub struct GreetingAccount {
   pub name: String,
}
```

Make sure your file is saved.! We're now ready to build and deploy the program to your local Solana cluster

Building and Deploying the Program

9. Use the following command to build the program with anchor. You should see a similar output:

```
cargo-build-bpf child: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/sdk/bpf
gm-anchor/target/bpfel-unknown-unknown/release/gm_anchor.so /Users/pappas99/GitHub/solana-bootcamp/gm-anchor
cargo-build-bpf child: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/sdk/bpf
ls /Users/pappas99/GitHub/solana-bootcamp/gm-anchor/target/deploy/gm_anchor.so

To deploy this program:
  $ solana program deploy /Users/pappas99/GitHub/solana-bootcamp/gm-anchor/target/deploy/gm_anchor.so
The program address will default to this keypair (override with --program-id):
  /Users/pappas99/GitHub/solana-bootcamp/gm-anchor/target/deploy/gm_anchor-keypair.json
pappas99@Pappas app %
```

10. Now that the program has been built, we need to extract the generated program ID and insert it back into the 'declare_id' line in the Rust program. We can obtain the program ID using the following command:

```
solana address -k
./target/deploy/gm_anchor-keypair.json
```

```
pappas99@Pappas gm-anchor % solana address -k ./target/deploy/gm_anchor-keypair.json
75PsQyHnFLhmF1jb4m31f2Gt35HFou3vJsaBCVAhYKAo
pappas99@Pappas gm-anchor %
```

11. Copy the program ID and paste it into the lib.rs program, replacing the string in the 'declare_id' line near the top of the program. This will tell Anchor what the program ID is so it can successfully connect and interact with the program once it's deployed. Save the file once.

```
declare_id!
("75PsQyHnFLhmF1jb4m31f2Gt35HFou3vJsaBCVAhYKAo");
```

12. Because you modified the Rust program, you need to now build it again

```
anchor build
```

13. Now we can deploy the program to the local cluster, using the following commands:

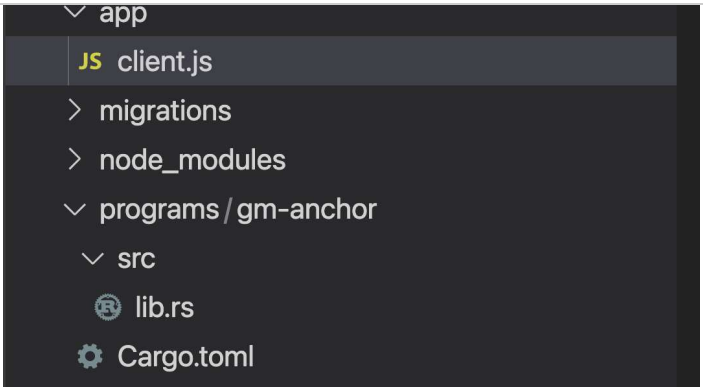
```
anchor deploy
```

If you're using the devnet network instead of a local validator, you may need to specifically state devnet:

```
anchor deploy --provider.cluster devnet
```

Congratulations, you just deployed your first Anchor Solana program! Now let's create a client to interact with it.

Creating the Client



15. Enter the following code into client.js. This client does the following:
    a. Takes two parameters from the command line:
        i. Program - the program ID of the deployed program
        ii. Name - the name we want to say GM to
    b. Uses the passed in program ID parameter, and the generated IDL file for the program, creates a connection to the deployed program
    c. Generates a new keypair to be used for storing the name data
    d. Calls the 'execute' instruction on the deployed program, passing in the account to store the name in, the user account paying for the transaction fees, and the system program ID. We also sign the transaction with the account storing the GM name
    e. Prints out the program log messages obtained once the transaction is confirmed
    f. Obtains the name stored in the account that was passed into the program, and prints the value to the console

Note: If you're using Devnet instead of localnet, change the word 'local' on line 2 to 'env'

```
const anchor = require("@project-serum/anchor");
const provider = anchor.Provider.local();
// Configure the cluster.
anchor.setProvider(provider);
const args = require('minimist')(process.argv.slice(2));

async function main() {
 // Read the generated IDL.
 const idl = JSON.parse(
   require("fs").readFileSync("../target/idl/gm_anchor.json",
"utf8")
 );

 // Address of the deployed program.
 const programId = new anchor.web3.PublicKey(args['program']);
 const name = args['name'] || "Glass Chewer";
```

```
 console.log('GM account public key: ' + gmAccount.publicKey);
 console.log('user public key: ' + provider.wallet.publicKey);

 // Execute the RPC.
 let tx = await program.rpc.execute(name,{
   accounts: {
     gmAccount: gmAccount.publicKey,
     user: provider.wallet.publicKey,
     systemProgram: anchor.web3.SystemProgram.programId
   },
   options: { commitment: "confirmed" },
   signers: [gmAccount],
 });

 console.log("Fetching transaction logs...");
 let t = await provider.connection.getConfirmedTransaction(tx,
"confirmed");
 console.log(t.meta.logMessages);
 // #endregion main

 // Fetch the account details of the account containing the price
data
 const storedName =
await program.account.greetingAccount.fetch(gmAccount.publicKey);
 console.log('Stored GM Name Is: ' + storedName.name)
}

console.log("Running client...");
main().then(() => console.log("Success"));
```

Running the Client

16. Anchor requires a couple of environment variables set to interact with deployed programs. Set the following environment variables on your terminal, to tell Anchor to use your previously created wallet account.Note if you're using the Devnet network instead of a local validator, set the ANCHOR_PROVIDER_URL to *https://api.devnet.solana.com*

```
export ANCHOR_PROVIDER_URL='http://127.0.0.1:8899'
```

    If you deployed to Devnet, you need to set it to the following:

```
export ANCHOR_PROVIDER_URL='https://api.devnet.solana.com'
```

17. You can run the client with the following commands, passing in the program and name parameters. You can put whatever you wish for the name:

```
cd app

node client.js --program $(solana address
-k ../target/deploy/gm_anchor-
keypair.json) --name Harry
```

```
  'Program 11111111111111111111111111111111 invoke [2]',
  'Program 11111111111111111111111111111111 success',
  'Program log: GM Harry',
  'Program 75PsQyHnFLhmF1jb4m31f2Gt35HFou3vJsaBCVAhYKAo consumed 12306 of 200000 compute units',
  'Program 75PsQyHnFLhmF1jb4m31f2Gt35HFou3vJsaBCVAhYKAo success'
]
Stored GM Name Is: Harry
```

Congratulations, you've successfully written your first solana program, as well as deployed it to a local Solana cluster and interacted with it with an off-chain client!

Completed code repository

Bonus Exercise:

You can attempt to complete these exercises if you've completed the main exercise ahead of schedule:

18. Modify your program and client so that it also stores a number that counts how many times the account has had GM said to it. To do this, you should initially set the state of the *number* variable to be 0, then increment it each time, storing it in the account.

Appendix: Errors

Error: *error[E0412]: cannot find type `ProgramResult` in this scope*

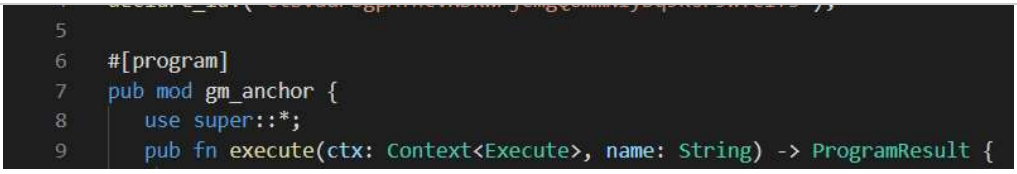
If you are running the latest version of anchor (e.g. `anchor-cli 0.23.0`, as of 4/1/2022) you may run into an error related to ProgramResult being undefined when running `anchor build`

```
BPF SDK: /home/will/.local/share/solana/install/releases/1.9.13/solana-release/bin/sdk/bpf
cargo-build-bpf child: rustup toolchain list -v
cargo-build-bpf child: cargo +bpf build --target bpfel-unknown-unknown --release
   Compiling solana-social v0.1.0 (/home/will/development/chainlink/solanaDeveloperBootcamp/solana-social/programs/solana-social)
error[E0412]: cannot find type `ProgramResult` in this scope
  --> programs/solana-social/src/lib.rs:24:84
   |
24 |     pub fn create_post(ctx: Context<CreatePost>, title: String, content: String) -> ProgramResult {
   |                                                                                      ^^^^^^^^^^^^^ not found in this scope

error[E0412]: cannot find type `ProgramResult` in this scope
  --> programs/solana-social/src/lib.rs:45:84
   |
45 |     pub fn update_post(ctx: Context<UpdatePost>, title: String, content: String) -> ProgramResult {
   |                                                                                      ^^^^^^^^^^^^^ not found in this scope

For more information about this error, try `rustc --explain E0412`.
error: could not compile `solana-social` due to 2 previous errors
```

To fix this, you'll just need to add the following line at the top of your `programs/gm-anchor/src/lib.rs` file:

```
use
anchor_lang::solana_program::entrypoint::ProgramResult;
```

Like this:



```
5
6   #[program]
7   pub mod gm_anchor {
8       use super::*;
9       pub fn execute(ctx: Context<Execute>, name: String) -> ProgramResult {
```

Once you add that line to `lib.rs`, just run `anchor build` again and you should be all set.

Error: *Deploying program failed: Error processing Instruction 1: custom program error: 0x1*
There was a problem deploying: Output { status: ExitStatus(unix_wait_status(256)), stdout: "", stderr: "" }.

If this error occurs when trying to deploy, just try to repeat the step

Exercise 2: Solana Social

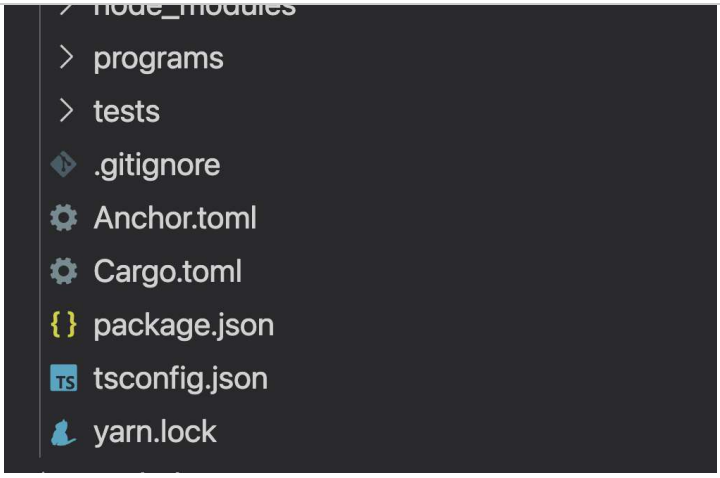
In this exercise we will create, deploy and interact with a solana program that stores and retrieves social media posts or blogs on-chain, using the Anchor framework

Initiating the project

1. The first step is to initiate a new project using the Anchor CLI. Ensuring you are in the folder that you want your exercise folder created, enter the following command into the terminal. It will create the necessary folders and files for your Anchor project. Once created, ensure you can see the folder structure in your VS Code folder explorer. In this example we enter 'cd ../..' first to go back two folders, as we were still in the app folder of our previous exercise:

```
cd ../..
anchor init solana-social
cd solana-social
```

   You should see a 'solana-social' folder created that contains all the necessary files and folders for your Anchor project. Ensure you can navigate this folder structure in the VS Code folder navigator (hint: if you need to find it, in VS Code go File → Open and find/choose the folder)



The next step is to install the minimist npm package. We will use it when we create the client later on

```
npm install minimist
npm install
```

2. Now we need to tell the Solana CLI that we want to use a local cluster. Note, If you're on Windows, it is recommended to use WSL to run these commands

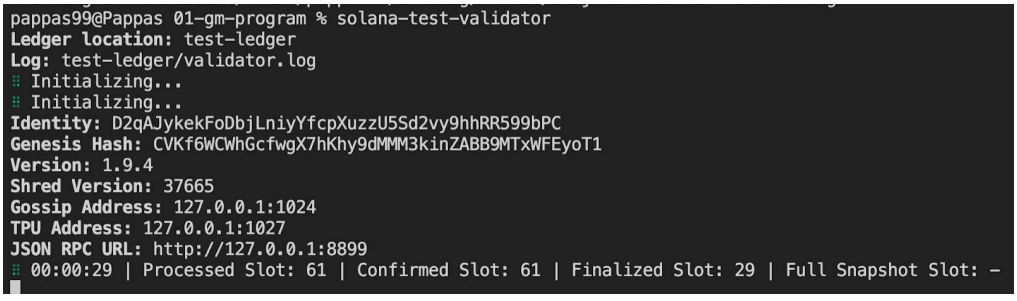
```
solana config set --url localhost
```

3. Now we can create a new CLI Keypair. We will use this for interacting with our local cluster.

```
solana-keygen new -o id.json
```

4. Next step is to start our local cluster. Note, you can skip this step if you still have a running local cluster from the previous exercise:

```
solana-test-validator
```



If you can't get the local validator running, then you can just use the public Devnet network. You can configure your setup as follows. Please do not run this command if you are able to successfully start the local validator.

creating transactions. We will do this by setting the ANCHOR_WALLET environment variable. If you are using windows, you may need to set these in your system environment variables. Note we're appending a '..' to it, because our client will be in a folder under the project root folder:

```
export ANCHOR_WALLET='../id.json'
```

5. Now that we have a new keypair, lets use the Solana airdrop program to retrieve some lamports that we can use to pay for transaction fees:

```
solana airdrop 2 $(solana-keygen pubkey
./id.json)
```

You're now ready to start building the on-chain program!

Creating the Program

In this section, we'll create a new Rust program that takes in a 'post' account, and then depending on if it's a 'create post' or 'update post' instruction, it will either store a social media post in the account, or it will update the values in a social media post.

6. The first step is to open the programs/solana-social/src/lib.rs file and enter in the following code. This code
    a. Defines a new anchor program
    b. Defines various constant variables to define the length of each field in a post, as well as the size of other fields to be stored in the post account
    c. Defines a new 'create_post' function, which takes an account from the context, and 'title' and ''content' parameters, validates the two input strings to ensure their length is valid, then creates the specified account with a new post containing the title, content, current timestamp and the author as the public key of the signing account
    d. Defines a new 'update_post' function, which takes an account from the context, and 'title' and ''content' parameters, validates the two input strings to ensure their length is valid, then updates the specified account with the new title, content and timestamp
    e. Creates an 'CreatePost' struct that defines the accounts passed into the create_post instruction, and the deserialization of the post account into a 'Post' struct

Solana Developer Bootcamp: Day 2 Exercises

Note: If you are using the latest version of Anchor (0.23) ,replace the text **ProgramResult** in the create_post and update_post function definitions (lines 24 and 45) with the text **Result<()>**, as the syntax has changed. In addition to this, you need to change [error] to [error_code] on line 88

You can check which version of Anchor you're running with the terminal command: *anchor --version*

```
use anchor_lang::prelude::*;
use anchor_lang::solana_program::system_program;

declare_id!
("BNDCEb5uXCuWDxJW9BGmbfvR1JBMAKckfhYrEKW2Bv1W");

const DISCRIMINATOR_LENGTH: usize = 8;
const PUBLIC_KEY_LENGTH: usize = 32;
const TIMESTAMP_LENGTH: usize = 8;
const STRING_LENGTH_PREFIX: usize = 4; // Stores
the size of the string.
const MAX_TITLE_LENGTH: usize = 100 * 4; // 50
chars max.
const MAX_CONTENT_LENGTH: usize = 500 * 4; // 280
chars max.

impl Post {
   const LEN: usize = DISCRIMINATOR_LENGTH
       + PUBLIC_KEY_LENGTH // Author.
       + TIMESTAMP_LENGTH // Timestamp.
       + STRING_LENGTH_PREFIX + MAX_TITLE_LENGTH
// Topic.
       + STRING_LENGTH_PREFIX +
MAX_CONTENT_LENGTH; // Content.
}

#[program]
pub mod solana_social {
   use super::*;
   pub fn create_post(ctx: Context<CreatePost>,
title: String, content: String) -> ProgramResult
{
       let post: &mut Account<Post> =
&mut ctx.accounts.post;
       let author: &Signer =
&ctx.accounts.author;
       let clock: Clock = Clock::get().unwrap();

       if title.chars().count() > 50 {

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

       if content.chars().count() > 280 {

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

       post.author = *author.key;
       post.timestamp = clock.unix_timestamp;
```

```
title: String, content: String) -> ProgramResult
{
       let post: &mut Account<Post> =
&mut ctx.accounts.post;

       if title.chars().count() > 50 {

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

       if content.chars().count() > 280 {

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

       post.title = title;
       post.content = content;

       Ok(())
   }
}

#[derive(Accounts)]
pub struct CreatePost<'info> {
   #[account(init, payer = author, space =
Post::LEN)]
   pub post: Account<'info, Post>,
   #[account(mut)]
   pub author: Signer<'info>,
   #[account(address = system_program::ID)]
   pub system_program: AccountInfo<'info>,
}

#[derive(Accounts)]
pub struct UpdatePost<'info> {
   #[account(mut, has_one = author)]
   pub post: Account<'info, Post>,
   pub author: Signer<'info>,
}

#[account]
pub struct Post {
   pub author: Pubkey,
   pub title: String,
   pub content: String,
   pub timestamp: i64,
}

#[error]
pub enum ErrorCode {
   #[msg("The provided title should be 50
characters long maximum.")]
   TitleLength,
   #[msg("The provided content should be 280
characters long maximum.")]
   ContentTooLong,
}
```

Make sure your file is saved.! We're now ready to build and deploy the program to your local

7. Use the following command to build the program with anchor. You should see a similar output:

```
anchor build
```

```
pappas Open folder in new window (cmd + click) ld
BPF SDK: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/sdk/bpf
cargo-build-bpf child: rustup toolchain list -v
cargo-build-bpf child: cargo +bpf build --target bpfel-unknown-unknown --release
   Compiling solana-social v0.1.0 (/Users/pappas99/GitHub/solana-bootcamp/solana-social/programs/solana-so
    Finished release [optimized] target(s) in 1.87s
cargo-build-bpf child: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/sdk/b
t/bpfel-unknown-unknown/release/solana_social.so /Users/pappas99/GitHub/solana-bootcamp/solana-social/targ
cargo-build-bpf child: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/sdk/b
GitHub/solana-bootcamp/solana-social/target/deploy/solana_social.so

To deploy this program:
  $ solana program deploy /Users/pappas99/GitHub/solana-bootcamp/solana-social/target/deploy/solana_social
The program address will default to this keypair (override with --program-id):
  /Users/pappas99/GitHub/solana-bootcamp/solana-social/target/deploy/solana_social-keypair.json
pappas99@Pappas solana-social %
```

8. Now that the program has been built, we need to extract the generated program ID and insert it back into the 'declare_id' line in the Rust program. We can obtain the program ID using the following command:

```
solana address -k
./target/deploy/solana_social-keypair.json
```

```
pappas99@Pappas solana-social % solana address -k ./target/deploy/solana_social-keypair.json
4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr
pappas99@Pappas solana-social %
```

9. Copy the program ID and paste it into the lib.rs program, replacing the string in the 'declare_id' line near the top of the program. This will tell Anchor what the program ID is so it can successfully connect and interact with the program once it's deployed. Save the file once.

```
declare_id!
("4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr");
```

10. Because you modified the Rust program, you need to now build it again

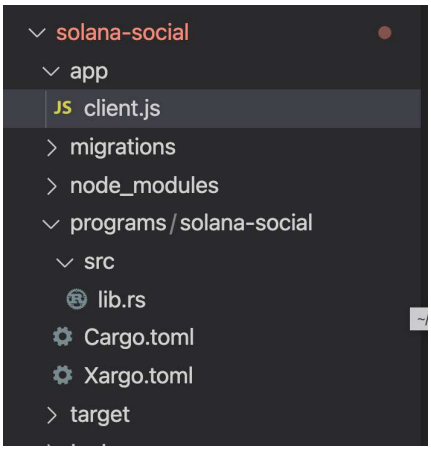
```
anchor build
```

11. Now we can deploy the program to the local cluster, using the following commands:

```
anchor deploy
```

Congratulations, you just deployed your second Anchor Solana program! Now let's create a client to interact with it.

Solana Developer Bootcamp: Day 2 Exercises



13. Enter the following code into client.js. This client does the following:
    a. Takes multiple parameters from the command line:
        i. Program - the program ID of the deployed program
        ii. Action - create or update a post
        iii. Title - the title of the blog post
        iv. Content - the content of the blog post
    b. Uses the passed in program ID parameter, and the generated IDL file for the program, creates a connection to the deployed program
    c. Generates a new keypair to be used for creating a new post
    d. Depending on the action parameter, it will call either the createPost or updatePost instruction, passing in the required accounts and string parameters
    e. Prints out the program log messages obtained once the transaction is confirmed
    f. Obtains the post data from the account that was passed into the program, and prints the values to the console

```
// Parse arguments
// --program - [Required] The account address for your deployed
program.
// --action - Create or Update a tweet
// --title - [Required] The title of the post
// --content - [Required] The content of the post
// --post - [Required] The account address of the post you wish
to update

const args = require('minimist')(process.argv.slice(2));


// Initialize Anchor and provider
const anchor = require("@project-serum/anchor");
const provider = anchor.Provider.env();
// Configure the cluster.
```

```
update)
const TITLE = args['title']  //title of the post
const CONTENT = args['content']  //content of the post
const POST_ACCT = args['post']

console.log('Program ID:', PROGRAM_ID);
console.log('Action: ' + ACTION);
console.log('user public key: ' + provider.wallet.publicKey);
console.log('creating post title: ' + TITLE, 'content: ' +
CONTENT);
console.log('post account: ' + POST_ACCT)


async function main() {
   // Read the generated IDL.
   const idl = JSON.parse(

 require("fs").readFileSync("../target/idl/solana_social.json",
"utf8")
   );


   // Generate the program client from IDL.
   const programId = new anchor.web3.PublicKey(PROGRAM_ID);
   const program = new anchor.Program(idl, programId);

   let tx

   if (ACTION === 'create') {
       // Create a Post
       //create an account to store the post
       const postAccount = anchor.web3.Keypair.generate();
       console.log('postAccount public key: ' +
postAccount.publicKey);

       tx = await program.rpc.createPost(TITLE, CONTENT, {
           accounts: {
               post: postAccount.publicKey,
               author: provider.wallet.publicKey,
               systemProgram:
anchor.web3.SystemProgram.programId
           },
           options: { commitment: "confirmed" },
           signers: [postAccount],
       });


       console.log("Fetching transaction logs...");
       let t =
await provider.connection.getConfirmedTransaction(tx,
"confirmed");
       console.log(t.meta.logMessages);

       // Fetch the account details of the account containing
the post
       const postData =
await program.account.post.fetch(postAccount.publicKey);
       console.log('Title Is: ' + postData.title)
       console.log('Content Is: ' + postData.content)
   }
```

```
            },
            options: { commitment: "confirmed" },
        });


        console.log("Fetching transaction logs...");
        let t =
await provider.connection.getConfirmedTransaction(tx,
"confirmed");
        console.log(t.meta.logMessages);

        // Fetch the account details of the account containing
the post
        const postData =
await program.account.post.fetch(POST_ACCT);
        console.log('Title Is: ' + postData.title)
        console.log('Content Is: ' + postData.content)

    } else {
        console.error('invalid action');
        return
    }

}

console.log("Running client...");
main().then(() => console.log("Success"));
```

Running the Client

14. Anchor requires a couple of environment variables set to interact with deployed programs. Set the following environment variables on your terminal, to tell Anchor to use your previously created wallet account. If you're using windows, you may need to set these in your windows environment variables.

```
export ANCHOR_WALLET='../id.json'
export ANCHOR_PROVIDER_URL='http://127.0.0.1:8899'
```

    Note: If you're using the public Devnet network instead of a local validator, set the ANCHOR_PROVIDER_URL to the devnet RPC endpoint:

```
export ANCHOR_PROVIDER_URL='https://api.devnet.solana.com'
```

15. You can run the client with the following command, passing in the program parameter, as well as the create action, then you can choose whatever you want for the post title and contents. This will create a new post for the specified post account that gets created:

```
cd app
```

```
r.json) --action update --title "this is a new title" --content "I have updated the post contents" --twe
et ADaNQkg9waXc16Nwq6jAd5VjxYPfJSwuY2AAdbUo2DHK
Program ID: 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr
pappas99@Pappas app % node client.js --program $(solana address -k ../target/deploy/solana_social-keypai
r.json) --action create --title "this is a title" --content "this is the content for the social media po
st"
Program ID: 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr
Action: create
user public key: HpnHTEgcLVxGUimVsqwbQUSZ24nHUuJogEdZ26Hzhhvt
creating post title: this is a title content: this is the content for the social media post
post account: undefined
Running client...
postAccount public key: 51hcKzuy1tdw5z6tt5dS4QPW5kUhKwEc1vh4HjMAQRLw
Fetching transaction logs...
[
  'Program 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr invoke [1]',
  'Program log: Instruction: CreatePost',
  'Program 11111111111111111111111111111111 invoke [2]',
  'Program 11111111111111111111111111111111 success',
  'Program 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr consumed 14821 of 200000 compute units',
  'Program 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr success'
]
Title Is: this is a title
Content Is: this is the content for the social media post
Success
pappas99@Pappas app %
```

16. Now that you've created a post, you can now run the client again with the `update` action to update the post. Be sure to put different values this time for the post title and contents. For the `post` parameter, you need to enter the **postAccount** public key that was outputted to the console in the previous step. This is so that we updated the correct account that was populated previously.

```
node client.js --program $(solana address
-k ../target/deploy/solana_social-
keypair.json) --action update --title
"this is a new title" --content "I have
updated the post contents" --post <insert
acct from previous tx>
```

```
pappas99@Pappas app % node client.js --program $(solana address -k ../target/deploy/solana_social-keypai
r.json) --action update --title "this is a new title" --content "I have updated the post contents" --pos
t 51hcKzuy1tdw5z6tt5dS4QPW5kUhKwEc1vh4HjMAQRLw
Program ID: 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr
Action: update
user public key: HpnHTEgcLVxGUimVsqwbQUSZ24nHUuJogEdZ26Hzhhvt
creating post title: this is a new title content: I have updated the post contents
post account: 51hcKzuy1tdw5z6tt5dS4QPW5kUhKwEc1vh4HjMAQRLw
Running client...
Fetching transaction logs...
[
  'Program 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr invoke [1]',
  'Program log: Instruction: UpdatePost',
  'Program 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr consumed 7545 of 200000 compute units',
  'Program 4BdMXdR5cgNTQiggcovRwTToydiiwXajdykiJn4VVVxr success'
]
Title Is: this is a new title
Content Is: I have updated the post contents
Success
```

Congratulations, you've successfully written, deployed and interacted with the solana social program!

Completed code repository

Bonus Exercise:

the contents of an account. Hint: you don't need to manually remove/clear the elements in the instruction logic, you can simply define the post account as follows in the DeletePost struct, and the 'close = ' will handle the removing/clearing of the account, and will send any remaining lamports back to the author

```
#[account(mut, has_one = author, close = author)]
```

Appendix: Errors

Error: error: cannot find attribute `error` in this scope

If you are running the latest version of anchor (e.g. `anchor-cli 0.23.0`, as of 4/1/2022) you may run into a variety of errors related to [error] having recently been changed to [error_code]

Example error output you may see:

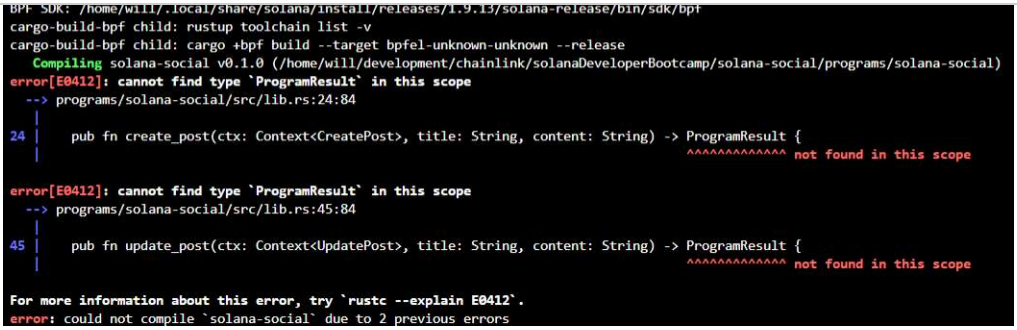
Solana Developer Bootcamp: Day 2 Exercises


```
                                                                                ^^^^^^^^^^^^^ not found in this scope

error[E0412]: cannot find type `ProgramResult` in this scope
  --> programs/solana-social/src/lib.rs:45:84
   |
45 |     pub fn update_post(ctx: Context<UpdatePost>, title: String, content: String) -> ProgramResult {
   |                                                                                      ^^^^^^^^^^^^^ not found in this scope

error[E0599]: no variant or associated item named `TitleLength` found for enum `anchor_lang::error::ErrorCode` in the current scope
  --> programs/solana-social/src/lib.rs:30:34
   |
30 |             return Err(ErrorCode::TitleLength.into())
   |                                   ^^^^^^^^^^^ variant or associated item not found in `anchor_lang::error::ErrorCode`

error[E0599]: no variant or associated item named `ContentTooLong` found for enum `anchor_lang::error::ErrorCode` in the current scope
  --> programs/solana-social/src/lib.rs:34:34
   |
34 |             return Err(ErrorCode::ContentTooLong.into())
   |                                   ^^^^^^^^^^^^^^ variant or associated item not found in `anchor_lang::error::ErrorCode`

error[E0599]: no variant or associated item named `TitleLength` found for enum `anchor_lang::error::ErrorCode` in the current scope
  --> programs/solana-social/src/lib.rs:49:34
   |
49 |             return Err(ErrorCode::TitleLength.into())
   |                                   ^^^^^^^^^^^ variant or associated item not found in `anchor_lang::error::ErrorCode`

error[E0599]: no variant or associated item named `ContentTooLong` found for enum `anchor_lang::error::ErrorCode` in the current scope
  --> programs/solana-social/src/lib.rs:53:34
   |
53 |             return Err(ErrorCode::ContentTooLong.into())
   |                                   ^^^^^^^^^^^^^^ variant or associated item not found in `anchor_lang::error::ErrorCode`

Some errors have detailed explanations: E0412, E0599.
For more information about an error, try `rustc --explain E0412`.
error: could not compile `solana-social` due to 9 previous errors
```

```
error: cannot find attribute `error` in this scope
  --> programs/solana-social/src/lib.rs:88:3
   |
88 | #[error]
   |   ^^^^^
   |
note: `error` is imported here, but it is a function-like macro
  --> programs/solana-social/src/lib.rs:1:5
   |
1  | use anchor_lang::prelude::*;
   |     ^^^^^^^^^^^^^^^^^^^^^^^

error: cannot find attribute `msg` in this scope
  --> programs/solana-social/src/lib.rs:90:6
   |
90 |     #[msg("The provided title should be 50 characters long maximum.")]
   |       ^^^
   |
note: `msg` is imported here, but it is a function-like macro
  --> programs/solana-social/src/lib.rs:1:5
   |
1  | use anchor_lang::prelude::*;
   |     ^^^^^^^^^^^^^^^^^^^^^^^

error: cannot find attribute `msg` in this scope
  --> programs/solana-social/src/lib.rs:92:6
   |
92 |     #[msg("The provided content should be 280 characters long maximum.")]
   |       ^^^
   |
note: `msg` is imported here, but it is a function-like macro
  --> programs/solana-social/src/lib.rs:1:5
   |
1  | use anchor_lang::prelude::*;
   |     ^^^^^^^^^^^^^^^^^^^^^^^

error[E0412]: cannot find type `ProgramResult` in this scope
  --> programs/solana-social/src/lib.rs:24:84
   |
24 |     pub fn create_post(ctx: Context<CreatePost>, title: String, content: String) -> ProgramResult {
   |                                                                                      ^^^^^^^^^^^^^ not found in this scope
```

If you run into this, you just need to change `[error]` to `[error_code]` on line #88 in lib.rs, here:

```
87
88  #[error_code]
89  pub enum ErrorCode {
90      #[msg("The provided title should be 50 characters long maximum.")]
91      TitleLength,
92      #[msg("The provided content should be 280 characters long maximum.")]
93      ContentTooLong,
94  }
```



To fix this, you'll just need to replace all occurrences of `ProgramResult` with `Result<()>` (on lines #24 and #45) in your `programs/solana-social/src/lib.rs` file.

Once you make those replacements in `lib.rs`, just run `anchor build` again and you should be all set.

Error: Error:
/Users/axelav/s/_scratch/solana-social/programs/solana-social/src/lib.rs:70:8
Struct field "system_program" is unsafe, but is not documented.
Please add a `/// CHECK:` doc comment explaining why no checks through types are necessary.
See https://book.anchor-lang.com/chapter_3/the_accounts_struct.html#safety-checks for more information.

Update the following code in your on-chain program:

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

To:

```
pub system_program: Program<'info,
System>,
```

And remove the macro on the line above:

```
#[account(address = system_program::ID)]
```

Exercise 3: Chainlink Price Feeds Consumer

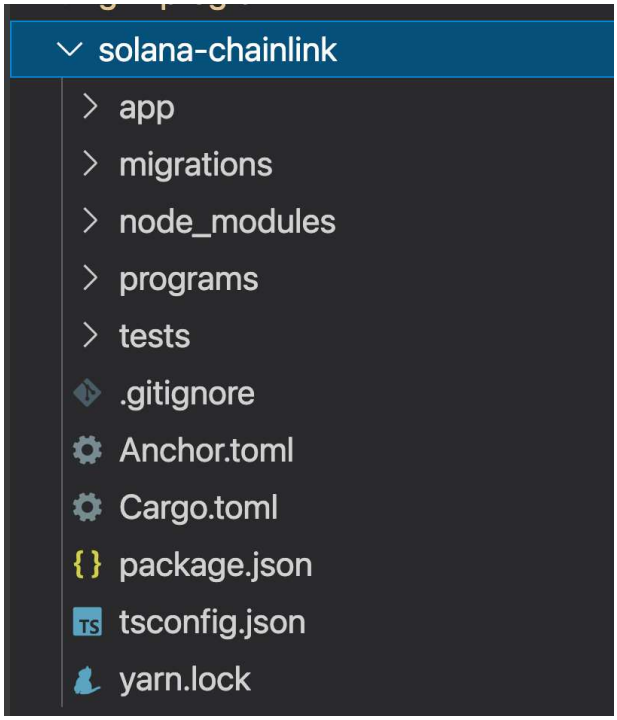
In this exercise we will create, deploy and interact with a solana program that will consume data from a Chainlink Data Feed. The program will take an account (to store the price data), and a Chainlink Data Feed

Initiating the project

1. First step is to initialize a new project using the Anchor CLI. Ensuring you are in the folder that you want your exercise folder created, enter the following command into the terminal. It will create the necessary folders and files for your Anchor project. Once created, ensure you can see the folder structure in your VS Code folder explorer. In this example we enter 'cd ..' first to go back a folder, as we were still in the folder of our previous exercise:

```
cd ../..
anchor init solana-chainlink
cd solana-chainlink
```

   You should see a 'solana-chainlink' folder created that contains all the necessary files and folders for your Anchor project. Ensure you can navigate this folder structure in the VS Code folder navigator (hint: if you need to find it, in VS Code go File → Open and find/choose the folder)



2. Now we need to tell the Solana CLI that we want to use the devnet cluster. Note, If you're on Windows, it is recommended to use WSL to run these commands

```
solana config set --url
https://api.devnet.solana.com
```

Solana airdrop program to retrieve some lamports that we can use to pay for transaction fees. In this instance, we're going to run the command twice to get 4 SOL worth of lamports:

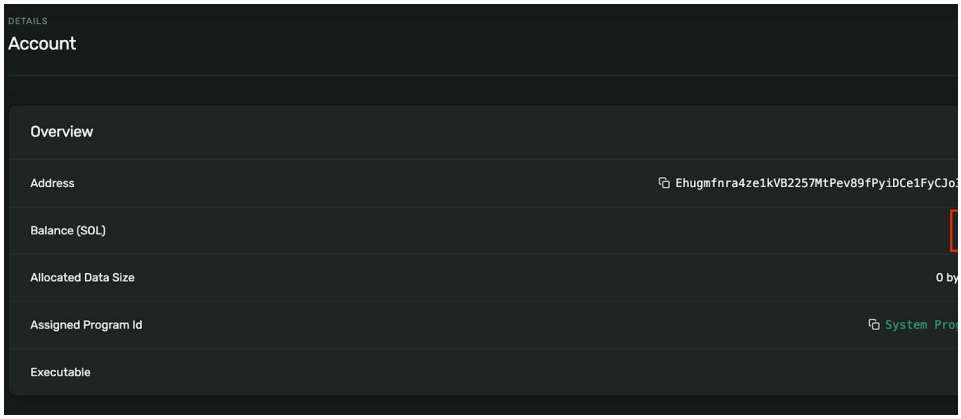
```
solana airdrop 2 $(solana-keygen pubkey
./id.json) && solana airdrop 2 $(solana-
keygen pubkey ./id.json)
```

5. You should be able to search for your public key on the Devnet explorer and see the SOL in your account. You can get your account public key with the following command:

```
solana-keygen pubkey id.json
```

```
pappas99@Pappas solana-chainlink % solana-keygen pubkey id.jsor
Ehugmfnra4ze1kVB2257MtPev89fPyiDCe1FyCJo3ipi
pappas99@Pappas solana-chainlink %
```

6. Then head to the Devnet explorer, and search for your account public key, and look for the SOL in your account. Ensure you have 4 SOL before continuing:



7. Now that we have a new keypair, let's also explicitly tell Anchor to use this keypair when creating transactions, and to tell it to use the public Devnet RPC endpoint for sending transactions. We will do this by opening the `Anchor.toml` file in the project root directory, and entering the following into the [**provider**] section, replacing anything already there:

```
cluster = "devnet"
wallet = "./id.json"
```

8. The final step is to add the Chainlink Solana crate to our project, so we can use Chainlink data feeds. Open the *Cargo.toml* file in the **programs/solana-chainlink** folder , and add this line to the dependencies section,

Creating the Program

In this section, we'll create a new Rust program that takes in a 'consumer' account, as well as a chainlink data feed account, and the account of the chainlink data feeds program, and it will obtain the latest price of the specified data feed, and store the result in the specified consumer account.

9. The first step is to open the /src/lib.rs file and enter in the following code in place of the existing code already there. This code:
    a. Defines a new anchor program
    b. Creates an Decimal struct that defines how the price data is stored in the specified consumer account
    c. Defines a fmt function for formatting price data with the correct decimals and zeros etc
    d. Defines a new 'execute' function, which takes the following as inputs:
        i. The consumer account to store the price data
        ii. The specified chainlink data feed account (eg SOL/USD) to obtain price data from
        iii. The chainlink price feeds program account on Devnet
    e. Defines the execute function body, which performs the following:
        i. Calls the `get_latest_round_data`, `description` and `decimals` functions of the chainlink data feed program for the specified price feed account
        ii. Stores the result of the calls above in the specified consumer account via the Decimal struct
        iii. Prints out the latest price to the program log output
    f. Defines the execute function context, and what accounts are expected as input when it's called

Note: If you are using the latest version of Anchor (0.23) ,replace the text **ProgramResult** in the execute function definition (line 39) with the text **Result<()>**, as the syntax has changed. You can check which version of Anchor you're running with the terminal command: *anchor --version*

```
use anchor_lang::prelude::*;
use anchor_lang::solana_program::system_program;

use chainlink_solana as chainlink;

declare_id!
("7Y7nxA4fDs1uWzFbjNiURHUe4Pcs9XJTyvrBufM6KE6F");
```

Solana Developer Bootcamp: Day 2 Exercises


```
impl Decimal {
   pub fn new(value: i128, decimals: u32) ->
Self {
       Decimal { value, decimals }
   }
}

impl std::fmt::Display for Decimal {
   fn fmt(&self, f: &mut std::fmt::Formatter<'_>)
-> std::fmt::Result {
       let mut scaled_val =
self.value.to_string();
       if scaled_val.len() <= self.decimals
as usize {
           scaled_val.insert_str(
               0,
               &vec!["0"; self.decimals as usize -
scaled_val.len()].join(""),
           );
           scaled_val.insert_str(0, "0.");
       } else {
           scaled_val.insert(scaled_val.len() -
self.decimals as usize, '.');
       }
       f.write_str(&scaled_val)
   }
}

#[program]
pub mod solana_chainlink {
   use super::*;
       pub fn execute(ctx: Context<Execute>) ->
ProgramResult  {
       let round = chainlink::latest_round_data(

 ctx.accounts.chainlink_program.to_account_info(),

 ctx.accounts.chainlink_feed.to_account_info(),
       )?;

       let description = chainlink::description(

 ctx.accounts.chainlink_program.to_account_info(),

 ctx.accounts.chainlink_feed.to_account_info(),
       )?;

       let decimals = chainlink::decimals(

 ctx.accounts.chainlink_program.to_account_info(),

 ctx.accounts.chainlink_feed.to_account_info(),
       )?;

       // Set the account value
       let decimal: &mut Account<Decimal> =
&mut ctx.accounts.decimal;
       decimal.value=round.answer;
       decimal.decimals=u32::from(decimals);
```

```
    }
}

#[derive(Accounts)]
pub struct Execute<'info> {
    #[account(init, payer = user, space = 100)]
    pub decimal: Account<'info, Decimal>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub chainlink_feed: AccountInfo<'info>,
    pub chainlink_program: AccountInfo<'info>,
    #[account(address = system_program::ID)]
    pub system_program: AccountInfo<'info>,
}
```

Note: If you are using the latest version of Anchor (0.23) ,replace the whole Execute struct definition (line 68 onwards),with the following. This is because the syntax has changed in the way the System program is referenced, and in how the newer version raises warnings if you don't have CHECK comments above accounts

You can check which version of Anchor you're running with the terminal command: *anchor --version*

```
pub struct Execute<'info> {
    #[account(init, payer = user, space =
100)]
    pub decimal: Account<'info, Decimal>,
    #[account(mut)]
    pub user: Signer<'info>,
    /// CHECK: We're reading data from this
specified chainlink feed
    pub chainlink_feed: AccountInfo<'info>,
    /// CHECK: This is the Chainlink
program library on Devnet
    pub chainlink_program:
AccountInfo<'info>,
    /// CHECK: This is the devnet system
program
    pub system_program: Program<'info,
System>,
}
```

Make sure your file is saved! We're now ready to build and deploy the program to the Devnet cluster.

Building and Deploying the Program

10. Use the following command to build the program with anchor. You should see a similar output:

```
anchor build
```

```
s /Users/pappas99/GitHub/solana-bootcamp/solana-chainlink/target/deploy/solana_chainlink.so

To deploy this program:
  $ solana program deploy /Users/pappas99/GitHub/solana-bootcamp/solana-chainlink/target/deploy/solana_chainlink.so
The program address will default to this keypair (override with --program-id):
  /Users/pappas99/GitHub/solana-bootcamp/solana-chainlink/target/deploy/solana_chainlink-keypair.json
pappas99@Pappas solana-chainlink %
```

11. Now that the program has been built, we need to extract the generated program ID and insert it back into the 'declare_id' line in the Rust program. We can obtain the program ID using the following command:

```
solana address -k
./target/deploy/solana_chainlink-
keypair.json
```

```
pappas99@Pappas solana-chainlink % solana address -k ./target/deploy/solana_chainlink-keypair.json
7Y7nxA4fDs1uWzFbjNiURHUe4Pcs9XJTyvrBufM6KE6F
pappas99@Pappas solana-chainlink %
```

12. Copy the program ID and paste it into the lib.rs program, replacing the string in the 'declare_id' line near the top of the program. This will tell Anchor what the program ID is so it can successfully connect and interact with the program once it's deployed. Save the file once you have finished modifying it

```
declare_id!
("7Y7nxA4fDs1uWzFbjNiURHUe4Pcs9XJTyvrBufM6KE6F");
```

13. Because you modified the Rust program, you need to now build it again

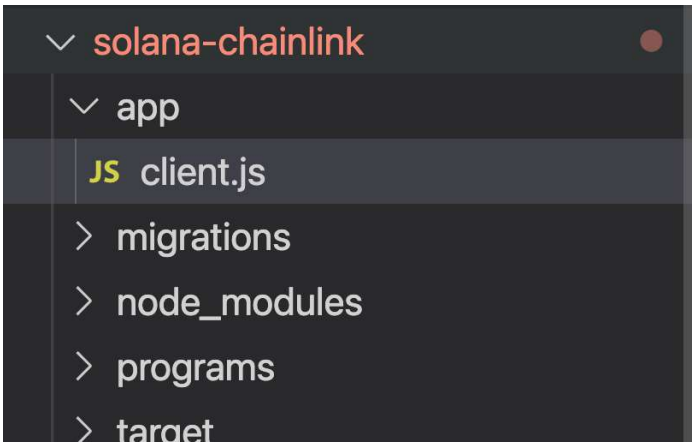
```
anchor build
```

14. Now we can deploy the program to the devnet cluster, using the following commands:

```
anchor deploy --provider.cluster devnet
```

```
pappas99@Pappas solana-chainlink % anchor deploy --provider.cluster devnet
Deploying workspace: https://api.devnet.solana.com
Upgrade authority: ./id.json
Deploying program "solana-chainlink"...
Program path: /Users/pappas99/GitHub/solana-bootcamp/solana-chainlink/target/
Program Id: 7Y7nxA4fDs1uWzFbjNiURHUe4Pcs9XJTyvrBufM6KE6F

Deploy success
pappas99@Pappas solana-chainlink %
```

Congratulations, you just deployed your Chainlink Solana program! Now let's create a client to interact with it.



16. Enter the following code into client.js. This client does the following:
    a. Takes two parameters from the command line:
        i. Program - the program ID of our deployed program
        ii. Feed - the chainlink data feed account that we want to obtain price data from
    b. Uses the passed in program ID parameter, and the generated IDL file for the program, creates a connection to the deployed program
    c. Generates a new keypair to be used for storing the price data
    d. Calls the 'execute' instruction on the deployed program, passing in the account to store the price data in in, the user account paying for the transaction fees, the Chainlink data feed account address of the data feed we are obtaining price data from, the Chainlink Price Feeds program on Devnet, and the system program ID. We also sign the transaction with the account storing the price data
    e. Prints out the program log messages obtained once the transaction is confirmed
    f. Obtains the latest price data stored in the consumer account that was passed into the program, and prints the value to the console

```
// Parse arguments
// --program - [Required] The account address for your deployed
program.
// --feed - The account address for the Chainlink data feed to
retrieve
const args = require('minimist')(process.argv.slice(2));

// Initialize Anchor and provider
const anchor = require("@project-serum/anchor");
const provider = anchor.Provider.env();
// Configure the cluster.
```

```
// Default is SOL / USD
const default_feed =
"EdWr4ww1Dq82vPe8GFjjcVPo2Qno3Nhn6baCgM3dCy28";
const CHAINLINK_FEED = args['feed'] || default_feed;

async function main() {
 // Read the generated IDL.
 const idl = JSON.parse(

 require("fs").readFileSync("../target/idl/solana_chainlink.json",
"utf8")
 );

 // Address of the deployed program.
 const programId = new anchor.web3.PublicKey(args['program']);

 // Generate the program client from IDL.
 const program = new anchor.Program(idl, programId);

 //create an account to store the price data
 const priceFeedAccount = anchor.web3.Keypair.generate();

 console.log('priceFeedAccount public key: ' +
priceFeedAccount.publicKey);
 console.log('user public key: ' + provider.wallet.publicKey);

 // Execute the RPC.
 let tx = await program.rpc.execute({
   accounts: {
     decimal: priceFeedAccount.publicKey,
     user: provider.wallet.publicKey,
     chainlinkFeed: CHAINLINK_FEED,
     chainlinkProgram: CHAINLINK_PROGRAM_ID,
     systemProgram: anchor.web3.SystemProgram.programId
   },
   options: { commitment: "confirmed" }, //start with the most
recent block that's confirmed
   signers: [priceFeedAccount],
 });

 console.log("Fetching transaction logs...");
 let t = await provider.connection.getConfirmedTransaction(tx,
"confirmed");
 console.log(t.meta.logMessages);

 // Fetch the account details of the account containing the price
data
 const latestPrice =
await program.account.decimal.fetch(priceFeedAccount.publicKey);
 console.log('Price Is: ' + latestPrice.value / DIVISOR)
}

console.log("Running client...");
main().then(() => console.log("Success"));
```

17. Install all the other required dependencies used in the client. This includes installing the minimist npm package, used for parsing command line arguments:

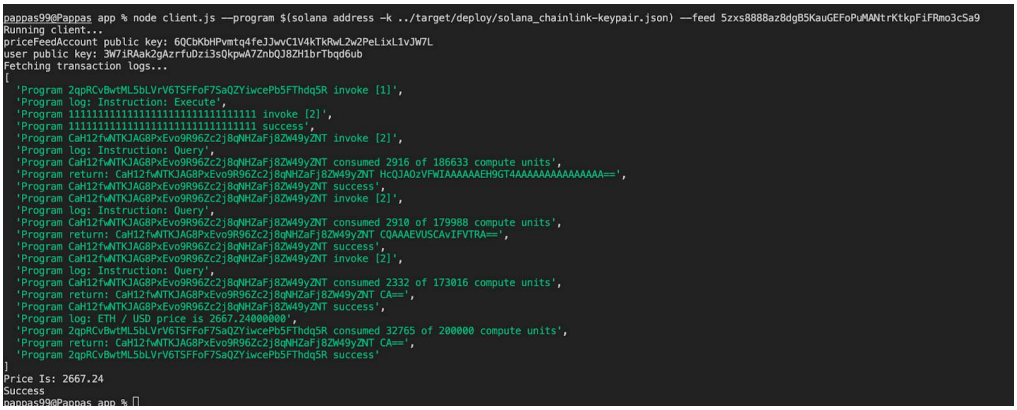
Running the Client

18. Anchor requires a couple of environment variables set to interact with deployed programs. Set the following environment variables on your terminal, to tell Anchor to use your previously created wallet account, and the public Devnet RPC URL when interacting with deployed programs:

```
export ANCHOR_PROVIDER_URL='https://api.devnet.solana.com'
export ANCHOR_WALLET='../id.json'
```

19. You can now run the client with the following commands, passing in the program and name parameters. You can choose whichever feed you like for the feed parameter

```
cd app

node client.js --program $(solana address -k
../target/deploy/solana_chainlink-
keypair.json) --feed
5zxs8888az8dgB5KauGEFoPuMANtrKtkpFiFRmo3cSa9
```



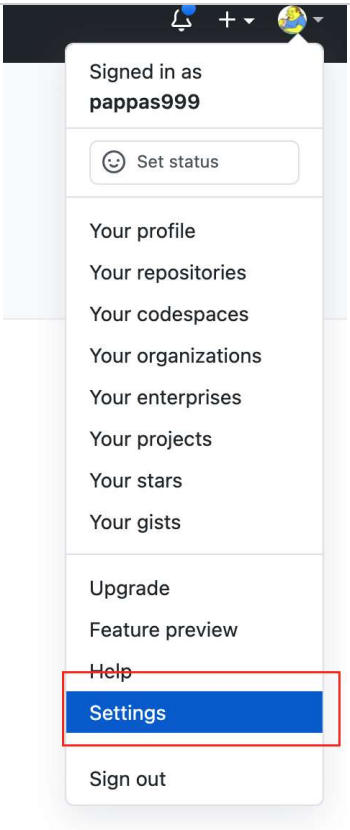
Congratulations, you've successfully written, deployed and interacted with a Solana program that uses Chainlink data feeds!

Completed code repository

Commit the repository to GitHub

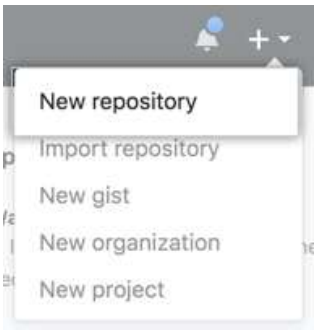
20. The final step is to commit your repository to GitHub and share your work! Perform the following steps to check in your Chainlink project into a public repository for everyone else to see your work

Solana Developer Bootcamp: Day 2 Exercises

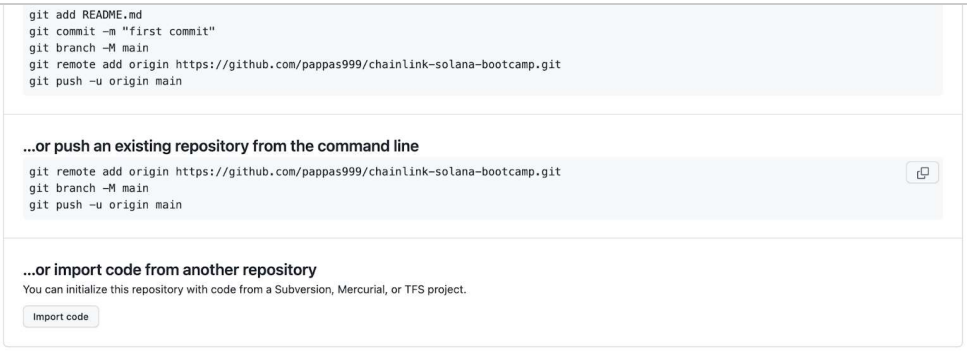


22. Generate a personal access token as per the GitHub instructions

23. Create a new repository on GitHub. Call it 'chainlink-solana-bootcamp', and leave the visibility as 'public'. To avoid errors, do not initialize the new repository with README, license, or gitignore files. Once you get to the 'setup page', leave it open, you will come back to it soon.



Solana Developer Bootcamp: Day 2 Exercises



24. Go back to your VS Code terminal for your completed project. Ensure you are in the project's root folder (cd ..) to go back a folder if you are still in the app folder

```
cd ..
```

25. Initialize the local directory as a Git repository.

```
git init -b main
```

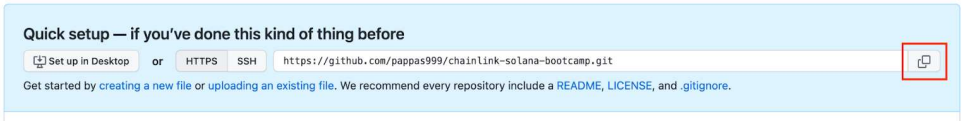
26. Add the files in your new local repository. This stages them for the first commit.
27.

```
git add .
```

28. Commit the files that you've staged in your local repository.

```
git commit -m "Solana Bootcamp".
```

29. Go back to GitHub. At the top of your GitHub repository's Quick Setup page, click the copy button to copy the remote repository URL.



30. Back in VS code terminal, you need to add the URL for the remote repository where your local repository will be pushed. Enter in the command below, and replace the <PASTE_REMOTE_URL> with the value you copied above. If prompted for GitHub authentication details, use your email that you signed up to GitHub with, and your personal access token that you generated earlier as the password. If you don't get prompted, you can continue.

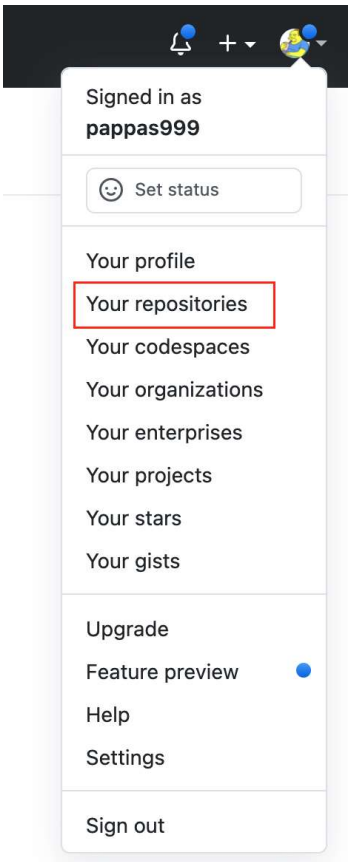
```
git remote add origin  <PASTE_REMOTE_URL>
```

```
pappas99@Pappas solana-chainlink % git remote -v
origin  https://github.com/pappas999/chainlink-solana-bootcamp.git (fetch)
origin  https://github.com/pappas999/chainlink-solana-bootcamp.git (push)
pappas99@Pappas solana-chainlink %
```

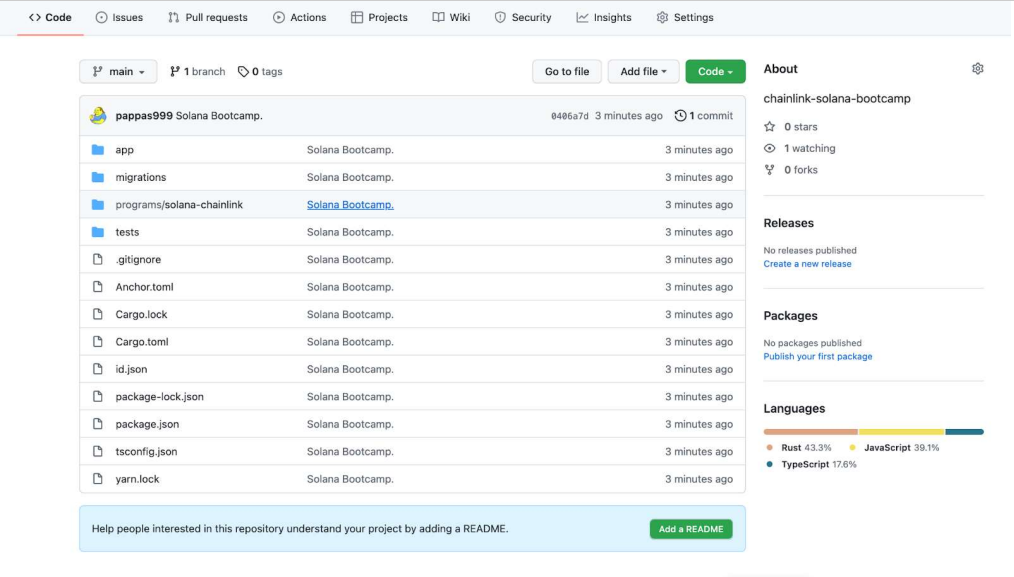
32. Push the changes in your local repository up to GitHub. If prompted for GitHub authentication details, use your email that you signed up to GitHub with, and your personal access token that you generated earlier as the password. This will give your local git program access to push code up to your GitHub account.

```
git push -u origin main
```

Your repository should now be live on GitHub! If you still have GitHub open with your new repository, then you can just refresh the page. Otherwise, head to GitHub, then in the top right corner open the menu and choose 'your repositories', then click on the URL link to your project to open it up and see. Take note of the URL and share it in the Bootcamp chat to share your completed project with everyone, or share it on Twitter or social media with the tag **#chainlink-solana-bootcamp**



Solana Developer Bootcamp: Day 2 Exercises



Bonus Exercise:

You can attempt to complete these exercises if you've completed the main exercise ahead of schedule:

33. Modify the program and client to take two feed parameters, and then use them both to create a new price feed. Eg ETH/USD and SOL/USD, then divide one by the other and return that result (in this case, it would be ETH/SOL)

Appendix: Errors

Error: invalid blockhash

If you get this error when trying to deploy to Devnet, try installing an older verison of the Solana CLI with the following command, as suggested in this stackoverflow post:

```
sh -c "$(curl -sSfL
https://release.solana.com/v1.8.13/install)"
```

Once this is done, try deploying your program again and it should work

Solana Developer Bootcamp: Day 2 Exercises