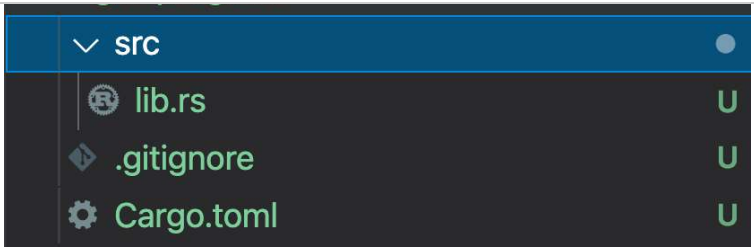
Solana Blockchain Developer Bootcamp Day 1 Exercises

Exercise 1: GM Smart Contract

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 will teach you about the basics of Solana programs and accounts, as well as serialization/deserialization.

Initiating the project

1. First step is to create a new project using cargo. Enter the following command into



2. The next step is to initiate a new project in that folder with npm using the following command in the terminal. You can accept all default values

```
cd gm-program
npm init -y
```

3. Next, we need to add all the required dependencies for our program. Replace the contents of your package.json file with the text below:

```
{
 "name": "01-gm-program",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "start": "ts-node ./client/main.ts",
   "start-with-test-validator": "start-
server-and-test 'solana-test-validator
```

```
dir=dist/program",
   "clean:program-rust": "cargo clean -
-manifest-path=./src/program-
rust/Cargo.toml && rm -rf ./dist"
 },
 "dependencies": {
   "@solana/buffer-layout": "^4.0.0",
   "@solana/web3.js": "^1.7.0",
   "borsh": "^0.7.0",
   "buffer": "^6.0.3",
   "mz": "^2.7.0",
   "yaml": "^1.10.2"
 },
 "devDependencies": {
   "@tsconfig/recommended": "^1.0.1",
   "@types/eslint": "^8.2.2",
   "@types/eslint-plugin-prettier":
"^3.1.0",
   "@types/mz": "^2.7.2",
   "@types/prettier": "^2.1.5",
   "@types/yaml": "^1.9.7",
   "@typescript-eslint/eslint-plugin":
"^4.6.0",
   "@typescript-eslint/parser":
"^4.6.0",
   "eslint": "^7.12.1",
   "eslint-config-prettier": "^6.15.0",
   "eslint-plugin-prettier": "^4.0.0",
   "prettier": "^2.1.2",
   "start-server-and-test": "^1.11.6",
   "ts-node": "^10.0.0",
   "typescript": "^4.0.5"
 },
 "engines": {
   "node": ">=14.0.0"
 },
 "author": "",
 "license": "ISC"
}
```

4. Next, we need to install all the listed dependencies above. Enter the following into the command line:

```
npm install
npm install -g ts-node
```

5. Next step is to configure the Cargo.toml file with the correct values. Replace the contents of the file with this:

```
[package]
name = "gm-program"
version = "0.1.0"
edition = "2021"
```

```
borsh = "0.9.1"
borsh-derive = "0.9.1"
solana-program = "=1.7.9"

[dev-dependencies]
solana-program-test = "=1.7.9"
solana-sdk = "=1.7.9"

[lib]
name = "gm_program"
crate-type = ["cdylib", "lib"]
```

6. Now we need to tell the Solana CLI that we want to use a local cluster.

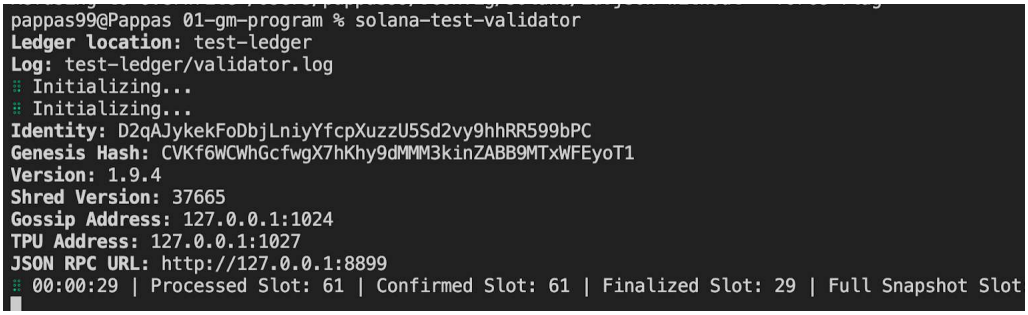
```
solana config set --url localhost
```

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

```
solana-keygen new
```

8. 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. If you still have the solana local validator running from the setup instructions, then you don't need to run this command

```
solana-test-validator
```



Now that you have a local cluster running, you can open a second terminal in VS code to continue entering CLI commands. You can do this with the '+' button near the top right of the terminal. You're now ready to start building the on-chain program

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
solana airdrop 2
```

You're now ready to start creating the 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 a new account which is then read by an off-chain client.

9. The first step is to open the /src/lib.rs file and enter in the following code. This code
    a. Tells the compiler we want to use Borsh for serializing and deserializing data
    b. Defines the program as a Solana program that takes the standard parameter inputs
    c. Creates a struct 'GreetingAccount' that will define how we read and store data in accounts
    d. Defines the entrypoint of the program as the process_instruction function
    e. Defines the skeleton of the process_instruction function

```
use borsh::{BorshDeserialize,
BorshSerialize};
use solana_program::{
   account_info::{next_account_info,
AccountInfo},
   entrypoint,
   entrypoint::ProgramResult,
   msg,
   program_error::ProgramError,
   pubkey::Pubkey,
};

/// Define the type of state stored in
accounts
#[derive(BorshSerialize,
BorshDeserialize, Debug)]
pub struct GreetingAccount {
   pub name: String,
}
```

```
    program_id: &Pubkey, // Public key
of the account the GM program was
loaded into
    accounts: &[AccountInfo], // The
account to say GM to
    input: &[u8], // String input data,
contains the name to say GM to
) -> ProgramResult {



    Ok(())
}
```

10. The final step is to flesh out the process_instruction function. Complete the function logic as per the text below. Be sure to save your file once it's modified. This function does the following
    a. Grabs the account that we want to store the GM name in
    b. Ensures the account can be written to
    c. Deserializes the input data from a byte array into a GreetingAccount struct
    d. Prints a message to the program output
    e. Serializes the GreetingAccount struct (turns it back into a byte array), and stores it in the passed in account

```
msg!("GM program entrypoint");

    // Iterating accounts is safer than indexing
    let accounts_iter = &mut accounts.iter();

    // Get the account to say GM to
    let account = next_account_info(accounts_iter)?;

    // The account must be owned by the program in order to
modify its data
    if account.owner != program_id {
        msg!("Greeted account does not have the correct program
id");
        return Err(ProgramError::IncorrectProgramId);
    }

    // Deserialize the input data, and store it in a
GreetingAccout struct
    let input_data =
GreetingAccount::try_from_slice(&input).unwrap();

    //Say GM in the Program output
```

Nice work! We're now ready to build and deploy the program to your local Solana cluster

Building and Deploying the Program

11. Use the following command to build the program. You should see a similar output:

```
cargo build-bpf --manifest-path=./Cargo.toml --bpf-out-dir=dist/program
```

```
pappas99@Pappas gm-program % cargo build-bpf --manifest-path=./Cargo.toml --bpf-out-dir=dist/program
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 gm-program v0.1.0 (/Users/pappas99/GitHub/solana-bootcamp/gm-program)
    Finished release [optimized] target(s) in 0.95s
cargo-build-bpf child: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/
t/bpfel-unknown-unknown/release/gm_program.so /Users/pappas99/GitHub/solana-bootcamp/gm-program/dist/
cargo-build-bpf child: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/
99/GitHub/solana-bootcamp/gm-program/dist/program/gm_program.so

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

12. Now we're ready to deploy our program to the localnet cluster:

```
solana program deploy dist/program/gm_program.so
```

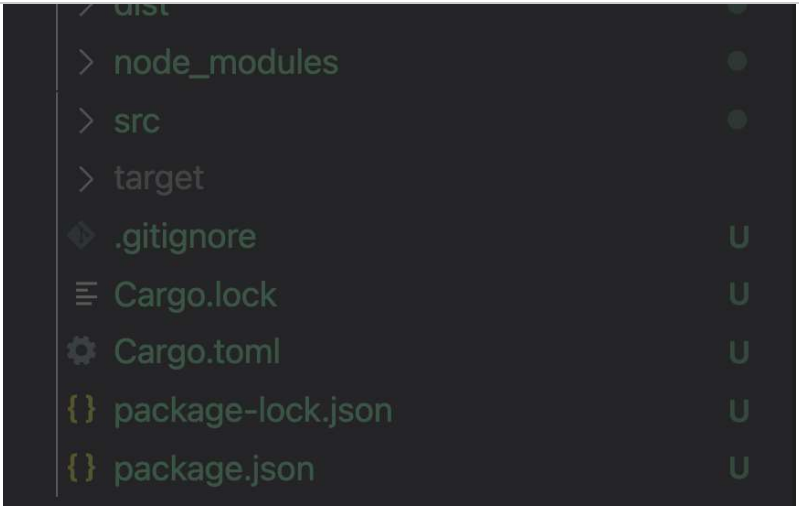
```
pappas99@Pappas gm-program % solana program deploy dist/program/gm_program.so
Program Id: 9e9KkhizPQVgEKuchWovFMxNcEwqNFhwRcN9DFBGQs2C

pappas99@Pappas gm-program %
```

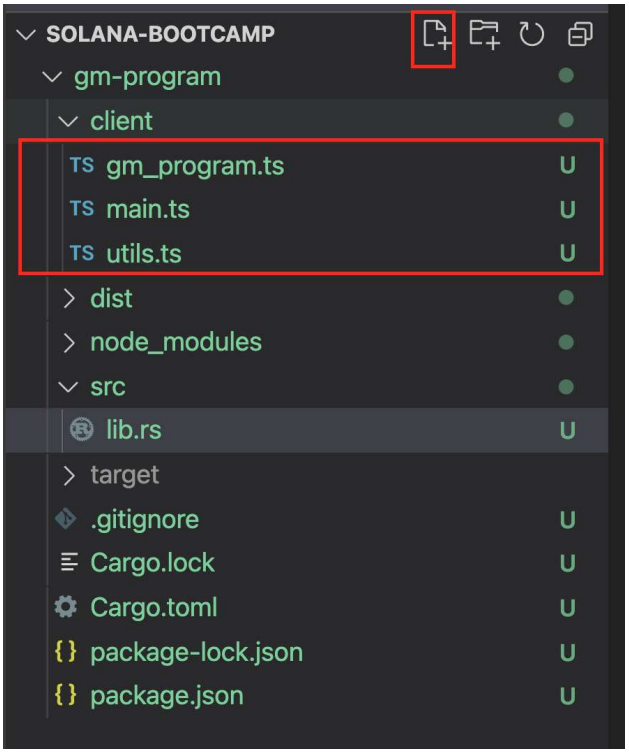
Congratulations, you just deployed your first Solana program! Now let's create a client to interact with it.

Creating the Client

13. First, let's create a new folder to store all our client code. Create a new folder in VS code, and call it 'client'



14. Inside the client folder, create the following files with the new file icon:
    a. gm_program.ts
    b. main.ts
    c. utils.ts



15. Enter the following code into main.ts. This will simply act as an entry point into the client, and then call all the functions required in the gm_programs file. Be sure to save your file once it's modified.

```
import {
```

```
async function main() {
  console.log("Let's say GM anon...");

  // Establish connection to the
cluster
  await establishConnection();

  // Determine who pays for the fees
  await establishPayer();

  // Check if the program has been
deployed
  await checkProgram();

  // Say hello to an account
  await sayGm();

  // Find out how many times that
account has been greeted
  await reportGm();

  console.log('Success');
}

main().then(
  () => process.exit(),
  err => {
    console.error(err);
    process.exit(-1);
  },
);
```

16. Next, enter the following code into utils.ts. These helper functions will be used to get config stored locally, get the RPC URL required to connect to a cluster, as well as generate a new keypair to be used for interacting with the deployed program. Be sure to save your file once it's modified.

    Note: If you aren't running a local validator and are deploying to the public Devnet, please replace the '**http://127.0.0.1:8899**' string in the getRpcUrl() function with '**https://api.devnet.solana.com**'

```
import os from 'os';
import fs from 'mz/fs';
import path from 'path';
import yaml from 'yaml';
import {Keypair} from '@solana/web3.js';

/**
* @private
*/
async function getConfig(): Promise<any> {
 // Path to Solana CLI config file
```

```
 const configYml = await fs.readFile(CONFIG_FILE_PATH,
{encoding: 'utf8'});
 return yaml.parse(configYml);
}

/**
* Load and parse the Solana CLI config file to determine
which RPC url to use
*/
export async function getRpcUrl(): Promise<string> {
   return 'http://127.0.0.1:8899';
}

/**
* Load and parse the Solana CLI config file to determine
which payer to use
*/
export async function getPayer(): Promise<Keypair> {
 try {
   const config = await getConfig();
   if (!config.keypair_path) throw new Error('Missing
keypair path');

 return await createKeypairFromFile(config.keypair_path);
 } catch (err) {
   console.warn(
     'Failed to create keypair from CLI config file,
falling back to new random keypair',
   );
   return Keypair.generate();
 }
}

/**
* Create a Keypair from a secret key stored in file as
bytes' array
*/
export async function createKeypairFromFile(
 filePath: string,
): Promise<Keypair> {
 const secretKeyString = await fs.readFile(filePath,
{encoding: 'utf8'});
 const secretKey =
Uint8Array.from(JSON.parse(secretKeyString));
 return Keypair.fromSecretKey(secretKey);
}
```

17. Lastly, enter the following code into your gm_program.ts. Be sure to save your file once it's modified. This is the main part of the client, and performs the following functionality:
    a. Establishes a connection to the cluster
    b. Establishes which account will be paying for the generated transactions

e. Reads the account data that was sent to the GM Program, and extracts the name that was stored in the account

```
import {
   Keypair,
   Connection,
   PublicKey,
   LAMPORTS_PER_SOL,
   SystemProgram,
   TransactionInstruction,
   Transaction,
   sendAndConfirmTransaction,
} from '@solana/web3.js';
import fs from 'mz/fs';
import path from 'path';
import * as borsh from 'borsh';
import { Buffer } from 'buffer';
import { getPayer, getRpcUrl, createKeypairFromFile }
from './utils';

/**
* Connection to the network
*/
let connection: Connection;

/**
* Keypair associated to the fees' payer
*/
let payer: Keypair;

/**
* Hello world's program id
*/
let programId: PublicKey;

/**
* The public key of the account we are saying hello to
*/
let greetedPubkey: PublicKey;

/**
* Path to program files
*/
const PROGRAM_PATH = path.resolve(__dirname, '../dist/program');

/**
* Path to program shared object file which should be deployed on
chain.
* This file is created when running either:
*   - `npm run build:program-c`
*   - `npm run build:program-rust`
*/
const PROGRAM_SO_PATH = path.join(PROGRAM_PATH, 'gm_program.so');

/**
```

Solana Developer Bootcamp: Day 1 Exercises


```
const NAME_FOR_GM='Glass Chewer'

/**
* Borsh class and schema definition for greeting accounts
*/

class GmAccount {
   name = "";
   constructor(fields: {name: string} | undefined = undefined) {
     if (fields) {
       this.name = fields.name;
     }
   }
   static schema = new Map([[GmAccount,
       {
           kind: 'struct',
           fields: [
               ['name', 'string']]
       }]]);
}


/**
* The expected size of each greeting account. Used for creating
the buffer
*/
const GREETING_SIZE = borsh.serialize(
   GmAccount.schema,
   new GmAccount({ name: NAME_FOR_GM }))
.length;

/**
* Establish a connection to the cluster
*/
export async function establishConnection(): Promise<void> {
   const rpcUrl = await getRpcUrl();
   connection = new Connection(rpcUrl, 'confirmed');
   const version = await connection.getVersion();
   console.log('Connection to cluster established:', rpcUrl,
version);
}

/**
* Establish an account to pay for everything
*/
export async function establishPayer(): Promise<void> {
   let fees = 0;
   if (!payer) {
       const { feeCalculator } =
await connection.getRecentBlockhash();

       // Calculate the cost to fund the greeter account
       fees +=
await connection.getMinimumBalanceForRentExemption(GREETING_SIZE);

       // Calculate the cost of sending transactions
       fees += feeCalculator.lamportsPerSignature * 100; // wag
```

```
request an airdrop
        const sig = await connection.requestAirdrop(
            payer.publicKey,
            fees - lamports,
        );
        await connection.confirmTransaction(sig);
        lamports = await connection.getBalance(payer.publicKey);
    }

    console.log(
        'Using account',
        payer.publicKey.toBase58(),
        'containing',
        lamports / LAMPORTS_PER_SOL,
        'SOL to pay for fees',
    );
}

/**
* Check if the hello world BPF program has been deployed
*/
export async function checkProgram(): Promise<void> {
    // Read program id from keypair file
    try {
        const programKeypair =
await createKeypairFromFile(PROGRAM_KEYPAIR_PATH);
        programId = programKeypair.publicKey;
    } catch (err) {
        const errMsg = (err as Error).message;
        throw new Error(
            `Failed to read program keypair at
'${PROGRAM_KEYPAIR_PATH}' due to error: ${errMsg}. Program may
need to be deployed with \`solana program deploy
dist/program/gm_program.so\``,
        );
    }

    // Check if the program has been deployed
    const programInfo = await connection.getAccountInfo(programId);
    if (programInfo === null) {
        if (fs.existsSync(PROGRAM_SO_PATH)) {
            throw new Error(
                'Program needs to be deployed with `solana program
deploy dist/program/gm_program.so`',
            );
        } else {
            throw new Error('Program needs to be built and
deployed');
        }
    } else if (!programInfo.executable) {
        throw new Error(`Program is not executable`);
    }
    console.log(`Using program ${programId.toBase58()}`);

    // Derive the address (public key) of a greeting account from
the program so that it's easy to find later.
    greetedPubkey = await PublicKey.createWithSeed(
```

Solana Developer Bootcamp: Day 1 Exercises


```
await connection.getAccountInfo(greetedPubkey);
   if (greetedAccount === null) {
       console.log(
           'Creating account',
           greetedPubkey.toBase58(),
           'to say hello to',
       );
       const lamports =
await connection.getMinimumBalanceForRentExemption(
           GREETING_SIZE,
       );

       const transaction = new Transaction().add(
           SystemProgram.createAccountWithSeed({
               fromPubkey: payer.publicKey,
               basePubkey: payer.publicKey,
               seed: NAME_FOR_GM,
               newAccountPubkey: greetedPubkey,
               lamports,
               space: GREETING_SIZE,
               programId,
           }),
       );
       await sendAndConfirmTransaction(connection, transaction,
[payer]);
   }
}

/**
* Say GM
*/
export async function sayGm(): Promise<void> {

   console.log('Saying hello to ',NAME_FOR_GM, ' with key ',
greetedPubkey.toBase58());

   //first we serialize the name data

   let gm = new GmAccount({
       name: NAME_FOR_GM
   })


   let data = borsh.serialize(GmAccount.schema, gm);
   const data_to_send = Buffer.from(data);
   console.log(data_to_send)

   const instruction = new TransactionInstruction({
       keys: [{ pubkey: greetedPubkey, isSigner: false,
isWritable: true }],
       programId,
       data: data_to_send
   });
   await sendAndConfirmTransaction(
       connection,
       new Transaction().add(instruction),
       [payer],
```

```
    const accountInfo =
await connection.getAccountInfo(greetedPubkey);
    if (accountInfo === null) {
        throw 'Error: cannot find the greeted account';
    }
    const greeting = borsh.deserialize(
        GmAccount.schema,
        GmAccount,
        accountInfo.data,
    );
    console.log(
        greetedPubkey.toBase58(),
        'GM was said to ',
        greeting.name
    );
}
```

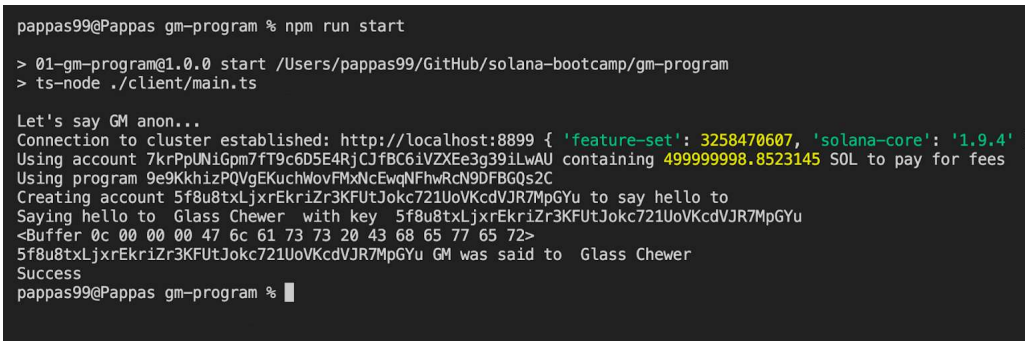
Running the Client

18. To run the client, first you can optionally set the value of the **NAME_FOR_GM** constant in the *gm_program.ts* file to be whatever name you wish. It's currently set to 'Glass Chewer'. Be sure to save your file once it's modified.

```
const NAME_FOR_GM='Glass Chewer'
```

19. Finally, you can now run your client. You should see output similar to the screenshot below

```
npm run start
```



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

20. 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. Feel free to reference the Solana Labs hello-world demo.

Exercise 2: Token Contract

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 will teach you about the basics of Solana programs and accounts, as well as serialization/deserialization.

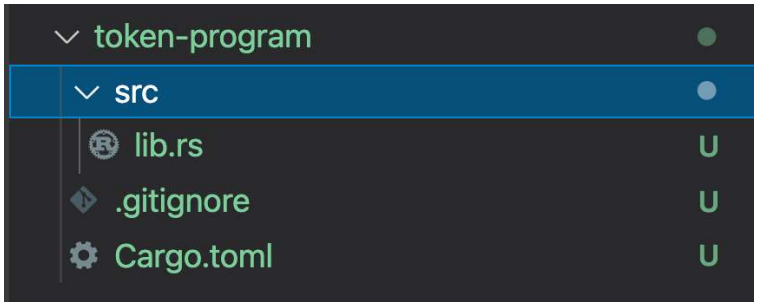
Initiating the project

1. First step is to create a new project using cargo. Enter the following command into the terminal in the folder containing all your projecst. It will create the necessary folders and files for the token program

   Optional: If you are still in the 'gm-program' folder, you will need to go back to a folder.

```
cd ..
```

```
cargo new token-program --lib
```



2. The next step is to initiate a new project in that folder with npm using the following

```
pappas99@Pappas solana-bootcamp % npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (solana-bootcamp)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/pappas99/GitHub/solana-bootcamp/package.json:

{
  "name": "solana-bootcamp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes)
```

3. Next, we need to add all the required dependencies for our program. Replace the contents of your package.json file with the text below

```
{
 "name": "01-token-program",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "start": "ts-node ./client/main.ts",
   "start-with-test-validator": "start-
server-and-test 'solana-test-validator
--reset --quiet'
http://localhost:8899/health start",
   "clean": "npm run clean:program-c &&
npm run clean:program-rust",
   "build:program-rust": "cargo build-
bpf --manifest-path=./src/program-
rust/Cargo.toml --bpf-out-
dir=dist/program",
   "clean:program-rust": "cargo clean -
-manifest-path=./src/program-
rust/Cargo.toml && rm -rf ./dist"
 },
 "dependencies": {
   "@solana/buffer-layout": "^4.0.0",
```

```
    "@tsconfig/recommended": "^1.0.1",
    "@types/eslint": "^8.2.2",
    "@types/eslint-plugin-prettier":
"^3.1.0",
    "@types/mz": "^2.7.2",
    "@types/prettier": "^2.1.5",
    "@types/yaml": "^1.9.7",
    "@typescript-eslint/eslint-plugin":
"^4.6.0",
    "@typescript-eslint/parser":
"^4.6.0",
    "eslint": "^7.12.1",
    "eslint-config-prettier": "^6.15.0",
    "eslint-plugin-prettier": "^4.0.0",
    "prettier": "^2.1.2",
    "start-server-and-test": "^1.11.6",
    "ts-node": "^10.0.0",
    "typescript": "^4.0.5"
  },
  "engines": {
    "node": ">=14.0.0"
  },
  "author": "",
  "license": "ISC"
}
```

4. Next, we need to install all the listed dependencies above. Enter the following into the command line:

```
npm install
```

21. Next step is to configure the Cargo.toml file with the correct values. Replace the contents of the file with this:

```
[package]
name = "token-program"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions
at https://doc.rust-
lang.org/cargo/reference/manifest.html
[features]
no-entrypoint = []

[dependencies]
borsh = "0.9.1"
borsh-derive = "0.9.1"
solana-program = "=1.7.9"
thiserror = "1.0"

[dev-dependencies]
solana-program-test = "=1.7.9"
solana-sdk = "=1.7.9"
```

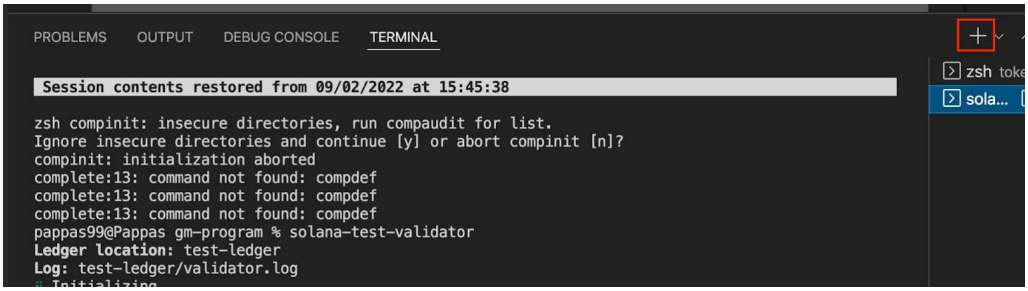
want to use a local cluster.

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

6. Now we can create a new CLI Keypair. We will use this for interacting with our local cluster. **Note** If you get an error stating 'Refusing to overwrite without –force flag', you can skip this step, as you're simply using the key created from the previous exercise.

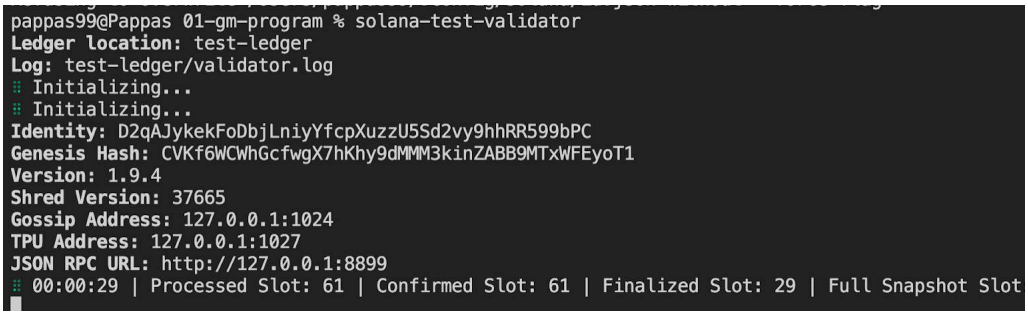
```
solana-keygen new
```

7. Because we want to run a local cluster in our existing VS Code terminal, you can open up a new terminal using the '+' button near the bottom right of VS Code, then use a second terminal for starting the valiator, leaving your first terminal for entering commands::



8. Next step is to start our local cluster. **This step can be skipped if you still have your local validator running from the previous exercise**. 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:

```
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

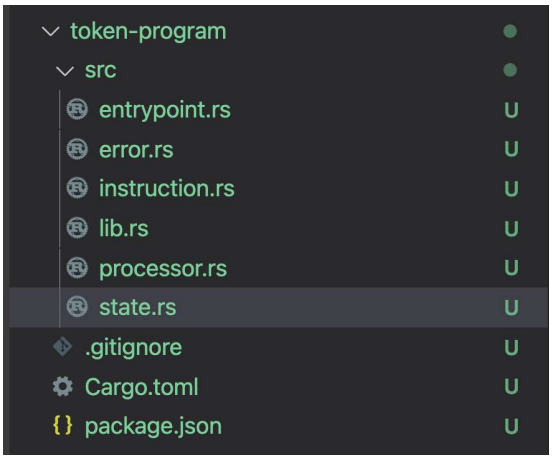
You're now ready to start creating the program

Creating the Program

In this section, we'll create a new Rust program that acts as a simplified version for the SPL Token Program. The SPL Token Program is a built-in Solana program that defines the standard for tokens on Solana.

9. The first step is to create the other required rust files in the 'src' folder. Using the new file icon, create the following files in the 'src' folder:
    a. entrypoint.rs
    b. instruction.rs
    c. processor.rs
    d. state.rs

The end result should be a folder and file structure like the image below:



10. The next step is to create the contents of the lib.rs file. This code will simply define all the modules required for the program. Each module is contained in it's own separate file:

```
pub mod entrypoint;
pub mod instruction;
pub mod processor;
pub mod state;
```

11. The next step is to populate the contents of the entrypoint.rs file. The code here performs the following:
    a. Defines the program as a Solana program
    b. Defines the entrypoint for the program as the

```
use crate::processor::Processor;

use solana_program::{
   account_info::{AccountInfo},
   entrypoint,
   entrypoint::ProgramResult,
   msg,
   pubkey::Pubkey,
};

// Declare and export the program's
entrypoint
entrypoint!(process_instruction);


fn process_instruction(
   program_id: &Pubkey,
   accounts: &[AccountInfo],
   instruction_data: &[u8],
) -> ProgramResult {
   msg!(
       "process_instruction: {}: {}
accounts, data={:?}",
       program_id,
       accounts.len(),
       instruction_data
   );


 Processor::process_instruction(program_id,
accounts, instruction_data)
}
```

12. The next step is to populate the contents of the instruction.rs file. The code here defines an enum that defines all the possible instructions that can be sent to the program:
    a. 0 - Create a new token
    b. 1 - Create a new token account
    c. 2 - Mint some tokens to a token account
    d. 3 - Transfer tokens between token accounts

    Take note that the Mint and Transfer values can contain an additional piece of data called ‘amount’

```
use borsh::{BorshDeserialize,
BorshSerialize};

#[derive(BorshSerialize,
```

13. The next step is to populate the contents of the state.rs file. The code here defines the data that will be stored for the program, which is split across two structs, a header level 'Token' struct that defines header level information about the token, as well as a 'TokenAccount' struct, that defines information about an account for the specified token. In addition to this, there are some helper functions included for each struct to help the program retrieve and save data

```
use borsh::{BorshDeserialize, BorshSerialize};
use solana_program::{
   account_info::AccountInfo,
entrypoint::ProgramResult,
program_error::ProgramError,
   pubkey::Pubkey,
};

#[derive(BorshSerialize, BorshDeserialize,
Debug, Clone)]
pub struct Token {
   pub authority: Pubkey,
   pub supply: u64,
}

impl Token {
   pub fn load_unchecked(ai: &AccountInfo) ->
Result<Self, ProgramError> {

 Ok(Self::try_from_slice(&ai.data.borrow())?)
   }
   pub fn save(&self, ai: &AccountInfo) ->
ProgramResult {

 Ok(self.serialize(&mut *ai.data.borrow_mut())?)
   }

   pub fn load(ai: &AccountInfo) -> Result<Self,
ProgramError> {
       let token =
Self::try_from_slice(&ai.data.borrow())?;
       Ok(token)
   }
}

#[derive(BorshSerialize, BorshDeserialize,
Debug, Clone)]
pub struct TokenAccount {
   pub owner: Pubkey,
   pub token: Pubkey,
   pub amount: u64,
}
```

```
ProgramResult {

 Ok(self.serialize(&mut *ai.data.borrow_mut())?)
   }

   pub fn load(ai: &AccountInfo) -> Result<Self,
ProgramError> {
       let account =
Self::try_from_slice(&ai.data.borrow())?;
       Ok(account)
   }
}
```

14. The final step is to populate the contents of the processor.rs file. The top section defines it as a Solana program, and tells the compiler we want to use the Borsh libraries, as well as the instruction and state modules that we created earlier. In addition to this, it defines a process_instruction function, which takes in the standard Solana Rust parameters, it retrieves the passed in instruction via the instruction_data parameter, then depending on what value it has (0,1,2,3), it calls different logic to perform the specified function. We'll fill in details for each instruction next

```
use borsh::{BorshDeserialize};
use solana_program::{
   account_info::{next_account_info, AccountInfo},
   entrypoint::ProgramResult,
   msg,
   program_error::ProgramError,
   pubkey::Pubkey,
};

use crate::instruction::TokenInstruction;
use crate::state::{Token, TokenAccount};

pub struct Processor {}

impl Processor {
   pub fn process_instruction(
       _program_id: &Pubkey,
       accounts: &[AccountInfo],
       instruction_data: &[u8],
   ) -> ProgramResult {
       let instruction =
TokenInstruction::try_from_slice(instruction_data)
           .map_err(|_|
ProgramError::InvalidInstructionData)?;
       let accounts_iter = &mut accounts.iter();
       msg!("Instruction: {:?}",instruction);
       match instruction {
           TokenInstruction::CreateToken => {
```

```
            }
            TokenInstruction::Mint { amount } => {
                msg!("Instruction: Mint");

            }
            TokenInstruction::Transfer { amount }
=> {
                msg!("Instruction: Transfer");
            }
        }
        Ok(())
    }
}
```

15. Now that we've defined the main body of our function, let's complete the logic for each branch of the match statement. The first is 'CreateToken', in this section we simply read in from the accounts array a new account to create a token for, as well as a 'token authority' account that acts as an owner of the token, and has authority to do things like mint new tokens etc. Then we simply set some default values for the token such as the supply, set the authority/owner to the passed in token authority account, then we save the token data into the passed in token master account.

```
                //get account info for master
token account
                let token_master_account =
next_account_info(accounts_iter)?;
                let token_authority =
next_account_info(accounts_iter)?;
                let mut token =
Token::load_unchecked(token_master_account)?;

                //set default values and save
master token account
                token.authority =
*token_authority.key;
                token.supply = 0;

 token.save(token_master_account)?
```

16. The next step is to fill out the logic for the 'CreateTokenAccount' section. The code takes in three accounts from the accounts array:
    a. New account to create a token account for
    b. The master token account that we want to create a token account under

```
                //get account info for master
token account and token account to be created
                let token_account_acct =
next_account_info(accounts_iter)?;
                let token_master_account =
next_account_info(accounts_iter)?;
                let owner =
next_account_info(accounts_iter)?;
                let mut token_account =
TokenAccount::load_unchecked(token_account_acct)?;

                //set default values and save token
account
                token_account.owner = *owner.key;
                token_account.token =
*token_master_account.key;
                token_account.amount = 0;

 token_account.save(token_account_acct)?
```

17. The next section to fill in is the Mint branch. In this part, the program looks for three accounts in the accounts array:
    a. The token account that wants to receive the minted tokens
    b. The master token account of the tokens that we want to mint
    c. The token authority account, that has access to mint new tokens

    The logic then does some basic validation to ensure that the passed in token authority account is the one that signed the transaction, otherwise it returns an error. After this check passes, it simply increases the total supply in the master token account, and then increases the balance of the token in the specified token account by the passed in value, and saves the state of the accounts

```
                //get account info for master
token account and token account to mint to
                let token_account_acct =
next_account_info(accounts_iter)?;
                let token_master_account =
next_account_info(accounts_iter)?;
                let mut token_account =
TokenAccount::load(token_account_acct)?;
                let mut token =
Token::load(token_master_account)?;

                //basic validation, ensure its the
master token authority trying to mint
                let token_authority =
next_account_info(accounts_iter)?;
```

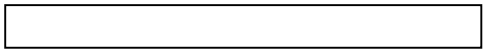
```
                //update total supply of the master
token, and update balance of token account that
received the mint
                token.supply += amount;
                token_account.amount += amount;

                //save updated contents of both
accounts

token_account.save(token_account_acct)?;
                token.save(token_master_account)?;
```

18. The final part of the logic that needs completing is the Transfer section. In this part of the program it looks for three accounts in the accounts array:
    a. The token account that is sending funds
    b. The token account receiving funds
    c. The owner of the token account sending funds

    The logic then performs some basic validation on the passed in accounts, ensuring the sender has enough funds, and that they are the one that signed the transaction. Once these checks pass, it simply updates the balances in the sender and receiver token accounts, and saves the new state



```
          //get account info for from and to token
accounts, as well as master token account
                let from_token_acct =
next_account_info(accounts_iter)?;
                let to_token_acct =
next_account_info(accounts_iter)?;
                let owner =
next_account_info(accounts_iter)?;
                let mut src_token_account =
TokenAccount::load(from_token_acct)?;
                let mut dst_token_account =
TokenAccount::load(to_token_acct)?;

                //basic validation, ensure sender has
enough funds
                if src_token_account.amount <= amount
{
                    msg!("Not enough tokens to
transfer");

 return Err(ProgramError::InsufficientFunds);
                }

                //ensure the owner of the from
```

```
                //ensure the owner passed in is the
actual owner of the token account
                if !(src_token_account.owner ==
*owner.key) {
                    msg!("Not the token account owner
signing the transaction");

 return Err(ProgramError::MissingRequiredSignature);
                }

                //update values in from and to
accounts, then save new contents of both accounts
                src_token_account.amount -= amount;
                dst_token_account.amount += amount;

 src_token_account.save(from_token_acct)?;

 dst_token_account.save(to_token_acct)?;
```

Awesome work! We're now ready to build and deploy the program to your local Solana cluster

Building and Deploying the Program

19. Use the following command to build the program. You should see a similar output:

```
cargo build-bpf --manifest-
path=./Cargo.toml --bpf-out-
dir=dist/program
```

```
pappas99@Pappas token-program % cargo build-bpf --manifest-path=./Cargo.toml --bpf-out-dir=dist/program
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
    Finished release [optimized] target(s) in 0.38s
cargo-build-bpf child: /Users/pappas99/.local/share/solana/install/releases/1.9.4/solana-release/bin/sdk/bpf/dependencies/bpf-tools/l
ols /Users/pappas99/GitHub/solana-bootcamp-walkthrough/token-program/dist/program/token_program.so

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

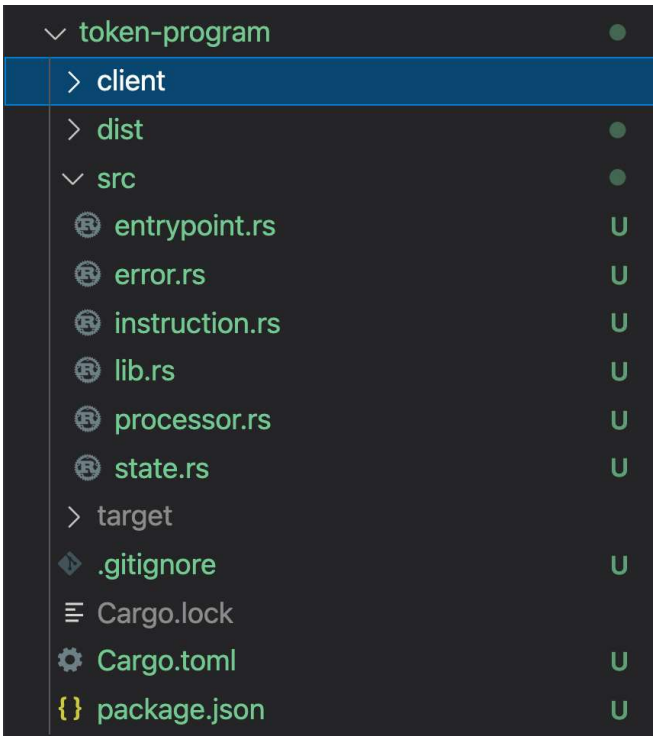
20. Now we're ready to deploy our program to the localnet cluster (or the Devnet public network if you're not running a local cluster):

```
solana program deploy
dist/program/token_program.so
```

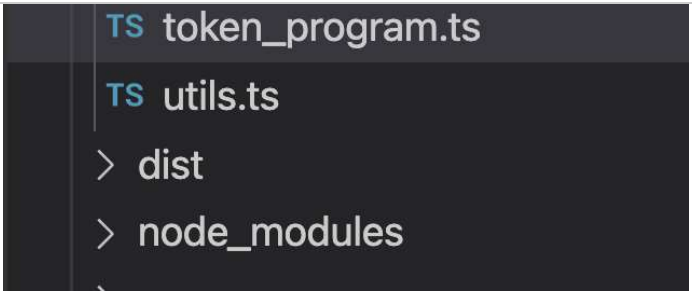
Great work, you've deployed your program to your local cluster! Now let's create a client to interact with it.

Creating the Client

21. First, let's create a new folder to store all our client code. Create a new folder in VS code under your 'token-program' folder, and call it 'client'



22. Inside the client folder, create the following files with the new file icon:
    a. token_program.ts
    b. main.ts
    c. utils.ts



23. Enter the following code into main.ts. This will simply act as an entry point into the client, and then call all the functions required in the token_program file. Be sure to save your file once it's modified.

```
import {
   establishConnection,
   establishPayer,
   checkProgram,
   createToken,
   createTokenAccounts,
   mint,
   transfer
 } from './token_program';

 async function main() {
   console.log("Let's create a
token...");

   // Establish connection to the
cluster
   await establishConnection();

   // Determine who pays for the fees
   await establishPayer();

   // Check if the program has been
deployed
   await checkProgram();

   // Create the master token
   await createToken();

   // Create two accounts to mint and
receive tokens
   await createTokenAccounts();

   // mint some tokens to one of the
two accounts
   await mint();

   // Send some tokens from the account
that recieved the mint to the second
account
   await transfer();
```

```
    console.error(err);
    process.exit(-1);
  },
);
```

24. Next, enter the following code into utils.ts. These helper functions will be used to get config stored locally, get the RPC URL required to connect to a cluster, as well as generate a new keypair to be used for interacting with the deployed program. Be sure to save your file once it's modified.

    Note: If you aren't running a local validator and are deploying to the public Devnet, please replace the '**http://127.0.0.1:8899**' string in the getRpcUrl() function with '**https://api.devnet.solana.com**'

```
import os from 'os';
import fs from 'mz/fs';
import path from 'path';
import yaml from 'yaml';
import {Keypair} from '@solana/web3.js';

/**
* @private
*/
async function getConfig(): Promise<any> {
 // Path to Solana CLI config file
 const CONFIG_FILE_PATH = path.resolve(
   os.homedir(),
   '.config',
   'solana',
   'cli',
   'config.yml',
 );
 const configYml = await fs.readFile(CONFIG_FILE_PATH,
{encoding: 'utf8'});
 return yaml.parse(configYml);
}

/**
* Load and parse the Solana CLI config file to determine
which RPC url to use
*/
export async function getRpcUrl(): Promise<string> {
   return 'http://127.0.0.1:8899';
}

/**
* Load and parse the Solana CLI config file to determine
which payer to use
*/
export async function getPayer(): Promise<Keypair> {
 try {
   const config = await getConfig();
```

```
falling back to new random keypair',
    );
    return Keypair.generate();
  }
}

/**
* Create a Keypair from a secret key stored in file as
bytes' array
*/
export async function createKeypairFromFile(
  filePath: string,
): Promise<Keypair> {
  const secretKeyString = await fs.readFile(filePath,
{encoding: 'utf8'});
  const secretKey =
Uint8Array.from(JSON.parse(secretKeyString));
  return Keypair.fromSecretKey(secretKey);
}
```

25. Lastly, enter the following code into your token_program.ts. Be sure to save your file once it's modified. This is the main part of the client, and performs the following functionality:
    a. Establishes a connection to the cluster
    b. Establishes which account will be paying for the generated transactions
    c. Ensures the token program has been deployed to the cluster
    d. Creates a new master token
    e. Creates two accounts and associated token accounts to be used for minting and sending tokens
    f. Mints some tokens to a specified token account
    g. Transfers some tokens from one token account to another

```
import {
    Keypair,
    Connection,
    PublicKey,
    LAMPORTS_PER_SOL,
    SystemProgram,
    TransactionInstruction,
    Transaction,
    sendAndConfirmTransaction,
} from '@solana/web3.js';

import fs from 'mz/fs';
import path from 'path';
import * as borsh from 'borsh';
import { Buffer } from 'buffer';
```

Solana Developer Bootcamp: Day 1 Exercises


```
/**
* Keypair associated to the fees' payer
*/
let payer: Keypair;

/**
* Token program id
*/
let programId: PublicKey;

/**
* The public key of the account that stores the token info
*/
let tokenPubkey: PublicKey;
let tokenFromAccountPubkey: PublicKey;
let tokenToAccountPubkey: PublicKey;
let tokenAccountPubkey: PublicKey;

/**
* Path to program files
*/
const PROGRAM_PATH = path.resolve(__dirname, '../dist/program');

/**
* Path to program shared object file which should be deployed on
chain.
* This file is created when running either:
*   - `npm run build:program-c`
*   - `npm run build:program-rust`
*/
const PROGRAM_SO_PATH = path.join(PROGRAM_PATH,
'token_program.so');

/**
* Path to the keypair of the deployed program.
* This file is created when running `solana program deploy
dist/program/gm_program.so`
*/
const PROGRAM_KEYPAIR_PATH = path.join(PROGRAM_PATH,
'token_program-keypair.json');

const TOKEN_NAME = 'GLASS_COIN'
const FROM_ACCT_SEED='FROM_ACCT_SEED'
const TO_ACCT_SEED='TO_ACCT_SEED'

/**
* Borsh class and schema definition for accounts
*/


class TokenInstruction {
   instruction = 0
   amount = 0
   constructor(fields: { instruction: number, amount: number } |
undefined = undefined) {
       if (fields) {
```

Solana Developer Bootcamp: Day 1 Exercises


```
            fields: [
                ['instruction', 'u8'],
                ['amount', 'u64']]
        }]]);
}

class TokenInstructionNoAmount {
    instruction = 0
    constructor(fields: { instruction: number } | undefined =
undefined) {
        if (fields) {
            this.instruction = fields.instruction;
        }
    }
    static schema = new Map([[TokenInstructionNoAmount,
        {
            kind: 'struct',
            fields: [
                ['instruction', 'u8']]
        }]]);
}

class Token {
    authority = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    supply = 0
    constructor(fields: { authority: [32], supply: number } |
undefined = undefined) {
        if (fields) {
            this.authority = fields.authority;
            this.supply = fields.supply;
        }
    }
    static schema = new Map([[Token,
        {
            kind: 'struct',
            fields: [
                ['authority', [32]],
                ['supply', 'u64']]
        }]]);
}


class TokenAccount {
    owner = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    token = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    amount = 0
    constructor(fields: { owner: [32], token: [32], amount: number }
| undefined = undefined) {
        if (fields) {
            this.owner = fields.owner;
            this.token = fields.token;
            this.amount = fields.amount;
        }
    }
```

Solana Developer Bootcamp: Day 1 Exercises


```
        }]]);
}

/**
* The expected size of each greeting account. Used for creating the
buffer
*/
const NEW_TOKEN_SIZE = borsh.serialize(
   Token.schema,
   new Token())
   .length;

const TOKEN_ACCOUNT_SIZE = borsh.serialize(
   TokenAccount.schema,
   new TokenAccount())
   .length;

/**
* Establish a connection to the cluster
*/
export async function establishConnection(): Promise<void> {
   console.log('getting connection')
   const rpcUrl = await getRpcUrl();
   connection = new Connection(rpcUrl, 'confirmed');
   const version = await connection.getVersion();
   console.log('Connection to cluster established:', rpcUrl,
version);
}

/**
* Establish an account to pay for creating the new token and
performing transactions
*/
export async function establishPayer(): Promise<void> {
   let fees = 0;
   if (!payer) {
       const { feeCalculator } =
await connection.getRecentBlockhash();

       // Calculate the cost to fund the greeter account
       fees +=
await connection.getMinimumBalanceForRentExemption(NEW_TOKEN_SIZE);

       // Calculate the cost of sending transactions
       fees += feeCalculator.lamportsPerSignature * 100; // wag

       payer = await getPayer();
   }

   let lamports = await connection.getBalance(payer.publicKey);
   if (lamports < fees) {
       // If current balance is not enough to pay for fees, request
an airdrop
       const sig = await connection.requestAirdrop(
           payer.publicKey,
           fees - lamports,
       );
```

Solana Developer Bootcamp: Day 1 Exercises


```
        'containing',
        lamports / LAMPORTS_PER_SOL,
        'SOL to pay for fees',
    );
}

/**
* Check if the token BPF program has been deployed
*/
export async function checkProgram(): Promise<void> {
    // Read program id from keypair file
    try {
        const programKeypair =
await createKeypairFromFile(PROGRAM_KEYPAIR_PATH);
        programId = programKeypair.publicKey;
    } catch (err) {
        const errMsg = (err as Error).message;
        throw new Error(
            `Failed to read program keypair at
'${PROGRAM_KEYPAIR_PATH}' due to error: ${errMsg}. Program may need
to be deployed with \`solana program deploy
dist/program/token_program.so\``,
        );
    }

    // Check if the program has been deployed
    const programInfo = await connection.getAccountInfo(programId);
    if (programInfo === null) {
        if (fs.existsSync(PROGRAM_SO_PATH)) {
            throw new Error(
                'Program needs to be deployed with `solana program
deploy dist/program/token_program.so`',
            );
        } else {
            throw new Error('Program needs to be built and
deployed');
        }
    } else if (!programInfo.executable) {
        throw new Error(`Program is not executable`);
    }
    console.log(`Using program ${programId.toBase58()}`);
    console.log('----------------------------------------------------
----------------------------------------------------------------')
}

/**
* Say GM
*/
export async function createToken(): Promise<void> {

    // First we'll check to see if the master token account has been
created aready, and if not we'll create it
    tokenPubkey = await PublicKey.createWithSeed(
        payer.publicKey,
        TOKEN_NAME,
        programId,
    );
```

Solana Developer Bootcamp: Day 1 Exercises


```
        console.log(
            'Creating account',
            tokenPubkey.toBase58(),
            'for our new token',
        );
        const lamports =
await connection.getMinimumBalanceForRentExemption(
            NEW_TOKEN_SIZE,
        );

        const transaction = new Transaction().add(
            SystemProgram.createAccountWithSeed({
                fromPubkey: payer.publicKey,
                basePubkey: payer.publicKey,
                seed: TOKEN_NAME,
                newAccountPubkey: tokenPubkey,
                lamports,
                space: NEW_TOKEN_SIZE,
                programId,
            }),
        );
        await sendAndConfirmTransaction(connection, transaction,
[payer]);


        console.log('Creating token ', TOKEN_NAME, ' with key ',
tokenPubkey.toBase58());

        // Create new master token
        //first we serialize the name data.
        let tokenInstruction = new TokenInstructionNoAmount({
"instruction": 0}) //instruction is 0 (create token)
        let data = borsh.serialize(TokenInstructionNoAmount.schema,
tokenInstruction);
        let dataBuffer = Buffer.from(data)

        //now we generate the instruction
        const instruction = new TransactionInstruction({
            keys: [
                { pubkey: tokenPubkey, isSigner: false, isWritable:
true },
                { pubkey: payer.publicKey, isSigner: false,
isWritable: false },
            ],
            programId,
            data: dataBuffer
        });
        await sendAndConfirmTransaction(
            connection,
            new Transaction().add(instruction),
            [payer],
        );

        console.log('Token successfully created at address ',
tokenPubkey.toBase58())

    } else {
```

Solana Developer Bootcamp: Day 1 Exercises


```
export async function createNewKeyPair(seed: string):
Promise<PublicKey> {

   console.log('Creating new keypair for seed: ', seed);

   //first we create the account and see if it exists already on-
chain
   tokenAccountPubkey = await PublicKey.createWithSeed(
       payer.publicKey,
       seed,
       programId,
   );

   //only create account if it doesn't already exist
   const tokenAcct =
await connection.getAccountInfo(tokenAccountPubkey);
   if (tokenAcct === null) {

       const lamportsTokenAccount =
await connection.getMinimumBalanceForRentExemption(
           TOKEN_ACCOUNT_SIZE,
       );

       //build up the instruction to create the account
       const transaction = new Transaction().add(
           SystemProgram.createAccountWithSeed({
               fromPubkey: payer.publicKey,
               basePubkey: payer.publicKey,
               seed: seed,
               newAccountPubkey: tokenAccountPubkey,
               lamports: lamportsTokenAccount,
               space: TOKEN_ACCOUNT_SIZE,
               programId,
           }),
       );
       await sendAndConfirmTransaction(
           connection,
           transaction,
           [payer],
       );

       console.log('created account for Public Key ',
tokenAccountPubkey.toBase58())


       //now that we've created the account, we can register is as
a token account
       await createTokenAccount(tokenAccountPubkey)

   } else {
       console.log('Token Account ', tokenAccountPubkey.toBase58(),
' already exists, skipping creation')
   }


   return tokenAccountPubkey
}
```

Solana Developer Bootcamp: Day 1 Exercises


```
tokenPubkey.toBase58());
   tokenFromAccountPubkey = await createNewKeyPair(FROM_ACCT_SEED)
   tokenToAccountPubkey = await createNewKeyPair(TO_ACCT_SEED)

   console.log('----------------------------------------------------
----------------------------------------------------------')
}

export async function createTokenAccount(tokenKey: PublicKey):
Promise<void> {

   //first we serialize the instruction data
   let tokenInstruction = new TokenInstructionNoAmount({
"instruction": 1 })  //1 = create token account
   let data = borsh.serialize(TokenInstructionNoAmount.schema,
tokenInstruction);
   let dataBuffer = Buffer.from(data)

   //now we build up the instruction
   const instruction = new TransactionInstruction({
       keys: [
           { pubkey: tokenKey, isSigner: false, isWritable: true },
           { pubkey: tokenPubkey, isSigner: false, isWritable:
true },
           { pubkey: payer.publicKey, isSigner: false, isWritable:
false }
       ],
       programId,
       data: dataBuffer
   });
   await sendAndConfirmTransaction(
       connection,
       new Transaction().add(instruction),
       [payer],
   );


   console.log('Token Account created for ', tokenKey.toBase58())
}



export async function mint(): Promise<void> {
   const MINT_AMOUNT = 100
   console.log('Minting ', MINT_AMOUNT, 'tokens of ', TOKEN_NAME, '
with key ', tokenPubkey.toBase58(), ' to account ',
tokenFromAccountPubkey.toBase58());

   //first we serialize the instruction data
   let tokenMintInstruction = new TokenInstruction({ "instruction":
2, "amount": MINT_AMOUNT })  //2 = mint tokens to account
   let data = borsh.serialize(TokenInstruction.schema,
tokenMintInstruction);
   let dataBuffer = Buffer.from(data)


   //now we build up the instruction
```

Solana Developer Bootcamp: Day 1 Exercises


```
false }
        ],
        programId,
        data: dataBuffer
    });
    await sendAndConfirmTransaction(
        connection,
        new Transaction().add(instruction),
        [payer],
    );


    console.log('getting account info for ',
tokenFromAccountPubkey.toBase58())
    await getAccountTokenInfo(tokenFromAccountPubkey)
    console.log('----------------------------------------------------
------------------------------------------------------------')
}


export async function transfer(): Promise<void> {
    const TRANSFER_AMOUNT = 5
    console.log('Transferring', TRANSFER_AMOUNT, 'of',
tokenPubkey.toBase58(), 'tokens from',
tokenFromAccountPubkey.toBase58(), 'to',
tokenToAccountPubkey.toBase58());

    //first we serialize the instruction data
    let tokenTransferInstruction = new TokenInstruction({
"instruction": 3, "amount": TRANSFER_AMOUNT })  //3 = transfer
tokens
    let data = borsh.serialize(TokenInstruction.schema,
tokenTransferInstruction);
    let dataBuffer = Buffer.from(data)


    //now we build up the instruction
    const instruction = new TransactionInstruction({
        keys: [
            { pubkey: tokenFromAccountPubkey, isSigner: false,
isWritable: true },
            { pubkey: tokenToAccountPubkey, isSigner: false,
isWritable: true },
            { pubkey: payer.publicKey, isSigner: false, isWritable:
false }
        ],
        programId,
        data: dataBuffer
    });
    await sendAndConfirmTransaction(
        connection,
        new Transaction().add(instruction),
        [payer],
    );
    await getAccountTokenInfo(tokenFromAccountPubkey)
    await getAccountTokenInfo(tokenToAccountPubkey)
    console.log('----------------------------------------------------
```

```
    const acct = await connection.getAccountInfo(account,
'processed');
    const data = Buffer.from(acct!.data);
    const accountInfo =
borsh.deserializeUnchecked(TokenAccount.schema, TokenAccount, data)
    console.log('account info for ', account.toBase58() + ': token
address:', new PublicKey(accountInfo.token).toBase58(), ',
balance:', accountInfo.amount.toString())
}
```

You're now ready to run the client!

Running the Client

26. To run the client, you simply use the pre-defined 'start' npm script in the package.json file:

```
npm run start
```

You should see all of the actions defined earlier in the program output. If you run the script a second time, you should see the minted tokens increase, as well as the transfer take place again.

Congratulations, you've successfully written, deployed and interacted with a token program!

[Completed code repository]

1. Add a ‘burn’ instruction to the program and client. The burn function should do the exact opposite of the ‘mint’ function, it should remove tokens from the authority account, and decrement the total supply. Remember to include the required checks necessary
2. Create a new SPL token using the actual SPL Token program with the Solana CLI

Appendix A:

Common errors:

Error during compilation on a Mac, especially after upgrading to Monterey: `xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun`

Solution: Install the command line tools pacakge