Lesson 11 - PDAs / Web3 continued

Cargo - Clippy tool

See Repo
See Docs
Clippy is a selection of linters for your rust code which gives you information in addition to the usual compiler warnings. It can guide you to best practices and more ideomatic code.
You can run it with

```
cargo clippy
```

It can also fix some problems, use

```
cargo clippy --fix
```

Interface Description Language (IDL)

This is a JSON file that provides information about our program in a similar way to an ABI file for solidity contracts.
It can be created when we build our program using Anchor, more of which tomorrow.

```
"version":"0.1.0","name":"hello_anchor","instructions":
[{"name":"initialize","accounts":
[{"name":"newAccount","isMut":true,"isSigner":true},
{"name":"signer","isMut":true,"isSigner":true},
{"name":"systemProgram","isMut":false,"isSigner":false}],"args":
[{"name":"data","type":"u64"}]}],"accounts":[{"name":"NewAccount","type":
{"kind":"struct","fields":[{"name":"data","type":"u64"}]}}]}
```

Upgrading Solana Programs

By default Solana programs can be modified and upgraded, in the Solana playground see the upgrade button once you have deployed your program.
This is achieved by the BPF loader which is the owner of every upgradable Solana program account.
There is a maximum limit to the size of the code.
Upgradability on blockchains is a means to do rug pulls, you should be cautious when taking this approach.

For more details see this article
The Upgradeable BPF loader program supports three different types of state accounts:

1. Program account: This is the main account of an on-chain program and its address is commonly referred to as a "program id." Program id's are what transaction instructions reference in order to invoke a program. Program accounts are immutable once deployed so you can think of them as a proxy account to the byte-code and state stored in other accounts.
2. Program data account: This account is what stores the executable byte-code of an on-chain program. When a program is upgraded, this account's data is updated with new byte-code. In addition to byte-code, program data accounts are also responsible for storing the slot when it was last modified and the address of the sole account authorised to modify the account (this address can be cleared to make a program immutable).
3. Buffer accounts: These accounts temporarily store byte-code while a program is being actively deployed through a series of transactions. They also each store the address of the sole account which is authorised to do writes.

Using the Solana CLI

We use the standard deploy command to re deploy.

```
solana program deploy <PROGRAM_FILEPATH>
```

By default, programs are deployed to accounts that are twice the size of the original deployment. Doing so leaves room for program growth in future redeployments.
But, if the initially deployed program is very small and then later grows substantially, the redeployment may fail.
To avoid this, specify a `max_len` that is at least the size (in bytes) that the program is expected to become .

```
solana program deploy --max-len 200000 <PROGRAM_FILEPATH>
```

PDA part 2

PDA creation process

1. Derive PDA (client)

   This javascript code will return PDA and a bump at which address didn't lay on the curve.

```
...

const [theAccountToInit, bump] = await PublicKey.findProgramAddress(
        [seed],
        programId       // intended owner
);

...
```

2. Assemble an instruction which includes necessary accounts (client)

   As the SystemProgram is invoked from within the program its account must also be provided.

```
...

const ix = new TransactionInstruction({
        keys: [
        {pubkey: payer.publicKey, isSigner: true, isWritable: true },
        {pubkey: theAccountToInit, isSigner: false, isWritable: true },
        {pubkey: SystemProgram.programId, isSigner: false,isWritable: false,},
        ],
        programId,      // program being called
        data: instruction_set, // Includes data on rent size
});

...
```

3. Call the program with the instructions (client)

```
...

await sendAndConfirmTransaction(
        connection,
        new Transaction().add(ix),
        [payer]
```

```
);

...
```

4. Create instruction for the System Program call (on-chain)

   At this stage you need to parse and pass who will pay for rent and how many lamports are needed to be rent exempt.

```
...

let ix = solana_program::system_instruction::create_account(
        funder.key,
        account_to_init.key,    // Account passed in the instruction
        lamports,
        account_size as u64,    // Account size retrieved from instruction_set
        program_id,
);

...
```

5. The Program Cross Program Invocation calls the System program with PDA creation instruction (on-chain)

   All of the arguments need to be provided and deserialised correctly.
   ...

```
invoke_signed(
        &ix,
        &[funder.clone(), account_to_init.clone()],
        &[&[seed.as_bytes(), &[bump]]],
)?;

...
```

Assuming the transaction went through, the program can now read and write to that account and other programs as well as clients can read its state.

Alternatively the PDA can be created on behalf of some program solely by the client:

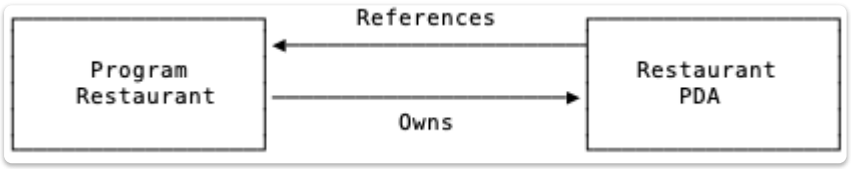
```
    ...

    let PDAPubKey = await PublicKey.createWithSeed(
```

```
                manager_account.publicKey,
                seed,
                programId
        );

        const transaction = new Transaction().add(
                SystemProgram.createAccountWithSeed({
                fromPubkey: payer.publicKey,
                basePubkey: payer.publicKey,
                seed: seed,
                newAccountPubkey: PDAPubKey,
                lamports: lamports,
                space: ACCESS_ACCOUNT_SIZE,
                programId: programId,
                })
        );

        await sendAndConfirmTransaction(connection, transaction,
[manager_account]);

        ...
```

Seed selection

Thought must be put into how to pick the source of the seed from which accounts will be derived.

Let's take an example of a program designed to provide restaurant booking and payment services in a web 3 manner, with core functionality residing inside the `Program Restaurant`.



A business wants to participate by adding themselves as another `Restaurant PDA` that would be owned by the program. What seed schema should the dApp architecture use to derive that PDA?

Some potential ideas for the seed management:

- Counter

  The seed could be an integer that gets incremented after each call.

  ```
  pda_s1 = findProgramDerivedAddress(programId, counter, seedBump)
  ```

  The 1st registration will use 0 for the seed, the 2nd registration will use 1 and so on.

  The drawback of this approach is that another account is needed to store the counter state

  The advantage is that we can iterate from 0-n on the client side, in order to check each of accounts for the one we want.

  As accounts are fixed in size and have a maximum size, PDAs can be used in a similiar manner to provide functionality of an ever growing list provided it doesn't exceed limits specified by the `counter` size which is indexing the list.

- Publickey

  The seed could be the public key of the wallet that someone at that business owns.

  ```
  pda_s2 = findProgramDerivedAddress(programId, businessWalletKey, seedBump)
  ```

  So each `Restaurant PDA` address will depend upon Publickey of whoever made the PDA creation call.

  Then, there is no need for a separate account that stores the state used for derivation.

  The drawback is that a caller would find it harder to iterate over all the potential `Restaurant Accounts` as Publickeys of all of the submitters would need to be known at the onset.

But anything can be used for the seed provided it makes sense within the application.

State chain

With initial seed sorted, further accounts can be derived from each other's addresses. An address of a PDA can be used as a seed component for another PDA.

These can be linked together to form a state chain allowing us to easily traverse along it just by knowing one exact state and the rule used to derive further references.

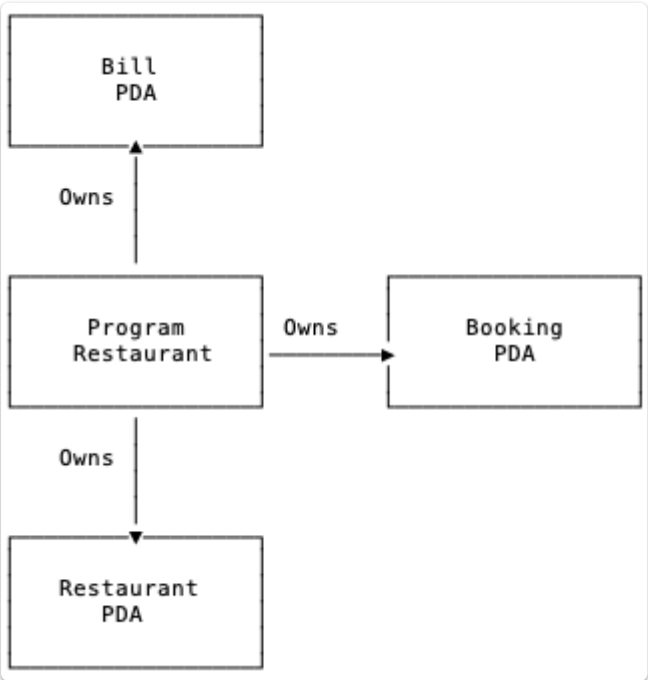
Each stage of that link would need its own seed derivation pattern.

To make a `Booking PDA` , a seed can also be a Publickey or a counter.
Using a Public key will restrict the caller to just a single booking, whilst using a counter will require iterating over `Booking PDA` for submissions from other people.

What can be done is to use a Public key and the counter as a seed!

```
seed = [restaurantPDAKey, restauranterPubKey, counter]
booking_pda = findProgramDerivedAddress(programId, seed, seedBump)
```

This means that for the client to use a `Program restaurant` they only need know the starting state and the logic necessary to get to the desired account, whether to read it or to modify it.



The above set-up would mean that the program owns and can modify each of the accounts (including transferring lamports), provided the caller has the authority to request such modification.

Seed schema correctness

Even after narrowing on the optimal seed source selection, a great care must be taken to ensure the scheme is exactly the same on both client and chain side to prevent this error:

Error: failed to send transaction: Transaction simulation failed: Error processing Instruction 0: Cross-program invocation with unauthorized signer or writable account

This error happens when our program communicates to the SystemProgram a desire to create an account that it actually can't sign for or that wasn't marked as writable.

The program has to invoke the System Program telling it what account it wants to create thus:

```
        invoke_signed(
                &ix,
                &[funder.clone(), account_to_init.clone()],
                &[&[seed.as_bytes(), &[bump]]],
        )?;
```

The client provides `account_to_init` and the `seed`.

SystemProgram will re derive the PDA and check:

- whether it is the same as `account_to_init.key` that was provided in the argument list and marks as writable
- whether the PDA was derived from the same Publickey as that of the program calling it

---

Web3 Further Details

Public Key

See Docs
PublicKey is used throughout `@solana/web3.js` in transactions, keypairs, and programs.
You require publickey when listing each account in a transaction and as a general identifier on Solana.

A PublicKey can be created with a base58 encoded string, buffer, Uint8Array, number, and an array of numbers.

```
const { Buffer } = require("buffer");
const web3 = require("@solana/web3.js");
const crypto = require("crypto");

// Create a PublicKey with a base58 encoded string
let base58publicKey = new web3.PublicKey(
  "5xot9PVkphiX2adznghwrAuxGs2zeWisNSxMW6hU6Hkj",
);
console.log(base58publicKey.toBase58());

// 5xot9PVkphiX2adznghwrAuxGs2zeWisNSxMW6hU6Hkj

// Create a Program Address
let highEntropyBuffer = crypto.randomBytes(31);
let programAddressFromKey = await web3.PublicKey.createProgramAddress(
  [highEntropyBuffer.slice(0, 31)],
  base58publicKey,
);
console.log(`Generated Program Address: ${programAddressFromKey.toBase58()}`);

// Generated Program Address: 3thxPEEz4EDWHNxo1LpEpsAxZryPAHyvNVXJEJWgBgwJ

// Find Program address given a PublicKey
let validProgramAddress = await web3.PublicKey.findProgramAddress(
  [Buffer.from("", "utf8")],
  programAddressFromKey,
);
console.log(`Valid Program Address: ${validProgramAddress}`);

// Valid Program Address: C14Gs3oyeXbASzwUpqSymCKpEyccfEuSe8VRar9vJQRE,253
```

Transactions continued

You can do multiple interactions within one transaction
Example from the documentation

```
const web3 = require("@solana/web3.js");
const nacl = require("tweetnacl");

// Airdrop SOL for paying transactions
let payer = web3.Keypair.generate();
let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

let airdropSignature = await connection.requestAirdrop(
  payer.publicKey,
  web3.LAMPORTS_PER_SOL,
);

await connection.confirmTransaction({ signature: airdropSignature });

let toAccount = web3.Keypair.generate();

// Create Simple Transaction
let transaction = new web3.Transaction();

// Add an instruction to execute
transaction.add(
  web3.SystemProgram.transfer({
    fromPubkey: payer.publicKey,
    toPubkey: toAccount.publicKey,
    lamports: 1000,
  }),
);

// Send and confirm transaction
// Note: feePayer is by default the first signer, or payer, if the parameter is not
await web3.sendAndConfirmTransaction(connection, transaction, [payer]);

// Alternatively, manually construct the transaction
let recentBlockhash = await connection.getRecentBlockhash();
let manualTransaction = new web3.Transaction({
  recentBlockhash: recentBlockhash.blockhash,
  feePayer: payer.publicKey,
});
manualTransaction.add(
  web3.SystemProgram.transfer({
    fromPubkey: payer.publicKey,
    toPubkey: toAccount.publicKey,
    lamports: 1000,
  }),
);


let transactionBuffer = manualTransaction.serializeMessage();
```

```
let transactionBuffer = manualTransaction.serializeMessage();
let signature = nacl.sign.detached(transactionBuffer, payer.secretKey);

manualTransaction.addSignature(payer.publicKey, signature);

let isVerifiedSignature = manualTransaction.verifySignatures();
console.log(`The signatures were verifed: ${isVerifiedSignature}`);

// The signatures were verified: true

let rawTransaction = manualTransaction.serialize();

await web3.sendAndConfirmRawTransaction(connection, rawTransaction);
```

System Program

Interacting with the system program allows us create accounts, allocate account data, assign an account to programs, work with nonce accounts, and transfer lamports.

```
const web3 = require("@solana/web3.js");

// Airdrop SOL for paying transactions
let payer = web3.Keypair.generate();
let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

let airdropSignature = await connection.requestAirdrop(
  payer.publicKey,
  web3.LAMPORTS_PER_SOL,
);

await connection.confirmTransaction({ signature: airdropSignature });

// Allocate Account Data
let allocatedAccount = web3.Keypair.generate();
let allocateInstruction = web3.SystemProgram.allocate({
  accountPubkey: allocatedAccount.publicKey,
  space: 100,
});
let transaction = new web3.Transaction().add(allocateInstruction);

await web3.sendAndConfirmTransaction(connection, transaction, [
  payer,
  allocatedAccount,
]);

// Create Nonce Account
let nonceAccount = web3.Keypair.generate();
let minimumAmountForNonceAccount =
  await connection.getMinimumBalanceForRentExemption(web3.NONCE_ACCOUNT_LENGTH);
let createNonceAccountTransaction = new web3.Transaction().add(
  web3.SystemProgram.createNonceAccount({
    fromPubkey: payer.publicKey,
    noncePubkey: nonceAccount.publicKey,
    authorizedPubkey: payer.publicKey,
    lamports: minimumAmountForNonceAccount,
  }),
);

await web3.sendAndConfirmTransaction(
  connection,
  createNonceAccountTransaction,
  [payer, nonceAccount],
);

// Advance nonce - Used to create transactions as an account custodian
let advanceNonceTransaction = new web3.Transaction().add(
```

```
  web3.SystemProgram.nonceAdvance({
    noncePubkey: nonceAccount.publicKey,
    authorizedPubkey: payer.publicKey,
  }),
);

await web3.sendAndConfirmTransaction(connection, advanceNonceTransaction, [
  payer,
]);

// Transfer lamports between accounts
let toAccount = web3.Keypair.generate();

let transferTransaction = new web3.Transaction().add(
  web3.SystemProgram.transfer({
    fromPubkey: payer.publicKey,
    toPubkey: toAccount.publicKey,
    lamports: 1000,
  }),
);
await web3.sendAndConfirmTransaction(connection, transferTransaction, [payer]);

// Assign a new account to a program
let programId = web3.Keypair.generate();
let assignedAccount = web3.Keypair.generate();

let assignTransaction = new web3.Transaction().add(
  web3.SystemProgram.assign({
    accountPubkey: assignedAccount.publicKey,
    programId: programId.publicKey,
  }),
);

await web3.sendAndConfirmTransaction(connection, assignTransaction, [
  payer,
  assignedAccount,
]);
```

Recent Blockhash and Nonce Accounts

See Docs
A transaction includes a recent blockhash to prevent duplication and to give transactions lifetimes. Any transaction that is completely identical to a previous one is rejected, so adding a newer blockhash allows multiple transactions to repeat the exact same action.
Transactions also have lifetimes that are defined by the blockhash, as any transaction whose blockhash is too old will be rejected.

An alternative approach is using nonce accounts.
Durable transaction nonces are a mechanism for getting around the typical short lifetime of a transaction's `recent_blockhash`. They are implemented as a Solana Program.

See Docs for more details.