TxPipe Docs

Quick Start

This guide will help you write your first Tx3 program. We'll create a simple transfer program that allows one party to send ADA to another.

We're assuming you have some experience building transactions for UTxO blockchains. If you need a refresher, here's a good one.

Note

Make sure you have followed the **installation steps** before you attempt to follow this guideline.

Initialize a new Project

The next thing you need to do is prepare a project workspace. This is a directory in your file system that holds everything you need for developing your protocol.

Lets create a `my-protocol` folder for our tutorial:

```
mkdir my-protocol && cd my-protocol
```

Once you're inside your protocol folder, you need to initialize the basic files using `trix`, the package manager for **tx3**.

```
trix init
```

The `init` command will command will ask you a few questions and then create a few files. Make sure to opt-in into the "Typescript" code generation option because we'll need it for the tutorial.

By then end, your protocol folder should look like:

```
my-protocol/
├── main.tx3
├── devnet.toml
└── trix.toml
```

Trix is to **Tx3** what **Npm** is to **NodeJS**

Your First Tx3 Protocol

The `trix init` command generates a `main.tx3` file with a very basic example. Open the file in your code editor of choice so that we can inspect the contents together.

TIP: if use VSCode, there's a Tx3 extension in the marketplace that provides many QoL features like syntax highlighting and error diagnostics.

```
party Sender;
party Receiver;

tx transfer(
    quantity: Int
) {
    input source {
        from: Sender,
        min_amount: Ada(quantity),
    }

    output {
        to: Receiver,
        amount: Ada(quantity),
    }

    output {
        to: Sender,
        amount: source - Ada(quantity) - fees,
    }
}
```

The above code represent a basic protocol with only one operation that transfers ADA from one party to another.

Lets break it down to understand each part.

Party Definitions

```
party Sender;
party Receiver;
```

Defines two parties that will participate in the transaction. These are placeholder names that will be bound to actual addresses at runtime.

In this particular scenario, we have a `Sender` that is transferring some ADA to a `Receiver` party.

You're not limited to only two parties, in more advanced scenarios you'll learn how to define any number of parties to fully describe your protocol.

Transaction Template

```
tx transfer(quantity: Int)
```

This snippet defines a transaction template named `transfer`. You can think of these templates as functions that your code will call to build concrete transactions.

This template takes a single parameter name `quantity` of type `Int`. In this particular scenario, this param determines how much ADA to transfer.

Inside this template you'll need to define how to connect inputs and outputs to fully describe your UTxO transaction.

Input Block

```
input source {
    from: Sender,
    min_amount: Ada(quantity),
}
```

This snippet defines an input block named `source`. An input block specifies the criteria that we need to use to query UTxO at runtime to fullfil the requirements of the transaction.

In this particular scenario we're requiring the input to come from the address belonging to the `Sender` and must contain at least the `quantity` of ADA specified as a parameter of the transaction.

Notice that we specified the name `source` to identify this particular input. Templates can have multiple input blocks. The name will be useful to reference this particular input from other parts of the transaction.

The `Ada(quantity)` syntax is an asset constructor. In more advanced scenarios you'll learn how to use other type of assets.

Output Block

```
output {
    to: Receiver,
    amount: Ada(quantity),
}
```

This snippet defines an output block. An output block describes how to construct UTxO that will be included at runtime as part of the concrete Tx.

In this particular scenario we're specifying that the UTxO should be locked in the address of the `Receiver` party and that the amount it should contain the exact `quantity` of ADA specified as a parameter.

Another Output Block

```
output {
    to: Sender,
    amount: source - Ada(quantity) - fees,
}
```

This snippet contains another output block. This time it represent the “change” we need to give back to the `Sender` party to balance the transaction.

Notice that the `amount` is now a computed value that takes whatever values was found in the `source` input and then subtracts the transferred value and the fees for the transaction.

The `fees` keyword is a special placeholder that will be replaced by the computed at runtime to match the minimum fees required for the concrete transaction being built.

Checking for Errors

The above code should be work out-of-the-box, but if you are building your own protocol you might want to check that everything compiles correctly.

The Tx3 compiler comes with a parser and semantic analysis logic to ensure that there aren’t any errors in your code.

To execute these checks, run the following command from the root of your protocol folder:

```
trix check
```

TIP: The VSCode extension comes with a live error diagnostics that use the same logic.

Running a Devnet

Tx3 tooling includes a mechanism to run a local ephemeral devnet. A devnet is a single-node network that runs on your machine and mimics the behavior of a real blockchain.

The node will emulate the chain moving forward by generating new blocks every few seconds. Any valid transaction submitted to the network will be automatically added to the following block.

It's *ephemeral* because any changes to the chain are reset after each restart, providing a nice deterministic workflow for testing transactions during development.

The config for you devnet lives inside the `devent.toml` file located in the root of your project.

```toml
[[utxos]]
address = "@alice"
value = 500000000

[[utxos]]
address = "@bob"
value = 500000000
```

Notice that the config allows you to define the initial UTxO set of the network. When the devnet starts, these UTxO will be part of the genesis block using the specified balances.

In this case, `@bob` and `@alice` are two dev wallets configured in your `trix.toml` file. You can create your initial UTxO pointing to these dev wallets by using the `@` prefix. Any other literal string will be interpreted as a bech32 address.

To start the devnet process, run the following command:

```
trix devnet
```

Once started, you should start seeing many logs in your temrinal. These logs describe the internal activity of the devnet.

TIP: the devnet mechanism is managed by Dolos, a lightweight Cardano Data Node.

Now that your devnet process is running, you can see what's happening by running the embedded chain explorer:

```
trix explore
```

What's next

If you've made it this far you should have a pretty good idea of what is Tx3 about, but this just barely scratched the surface.

Check the following section to continue your journey:

- Learn the details of the language in our Language Guide.
- Check a more complex scenario in our Example Catalog.
- Learn how to run an Ephemeral Devnet to try out your protocol.
- Learn how to install and use our VSCode extension.
- Understand the Tx3 Architecture making this work.