SubQuery

Making the world's decentralised data more accessible.

Lab Exercise Guide

Table of Contents

Introduction

In this exercise, we will take the starter project and focus on using an event handler to extract the balance of each account.

Pre-requisites

Completion of Module 1

Exercise 1: Account Balances

High level steps

1. Initialise the starter project
2. Update your mappings, manifest file and graphql schema file by removing all the default code except for the handleEvent function.
3. Generate, build and deploy your code
4. Deploy your code in Docker
5. Query for address balances in the playground

Detailed steps

Step 1: Initialise your project

The first step in creating a SubQuery project is to create a project with the following command:

```
$ subql init
Project name [subql-starter]: account-balance
? Select a network family Substrate
? Select a network Polkadot
? Select a template project subql-starter     Starter project for
subquery
RPC endpoint: [wss://polkadot.api.onfinality.io/public-ws]:
Git repository [https://github.com/subquery/subql-starter]:
Fetching network genesis hash... done
Author [Ian He & Jay Ji]:
Description [This project can be use as a starting po...]:
Version [1.0.0]:
License [MIT]:
Preparing project... done
account-balance is ready
```

Step 2: Update the graphql schema

The default schema.graphql file contains 5 fields. Rename field2 to account and field3 to balance. Rename the entity to Account.

> Extra: Whenever you update the manifest file, don't forget to update the reference to field1 in the mappings file appropriately and to regenerate the code via yarn codegen.

The schema file should look like this:

```
type Account @entity {
  id: ID! #id is a required field
  account: String #This is a Polkadot address
  balance: BigInt # This is the amount of DOT
}
```

Step 3: Update the manifest file (aka project.yaml)

The initialisation command also pre-creates a sample manifest file and defines 3 handlers. Because we are only focusing on Events, let's remove handleBlock and handleCall from the mappings file. The manifest file should look like this:

```
specVersion: 1.0.0
name: account-balance
version: 1.0.0
runner:
  node:
    name: '@subql/node'
    version: '>=1.0.0'
  query:
    name: '@subql/query'
    version: '*'
description: >-
  This project can be use as a starting point for developing your
SubQuery
  project
repository: 'https://github.com/subquery/subql-starter'
schema:
  file: ./schema.graphql
network:
  chainId:
'0x91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3'
  endpoint: 'wss://polkadot.api.onfinality.io/public-ws'
  dictionary:
'https://api.subquery.network/sq/subquery/polkadot-dictionary'
  #genesisHash:
```

```
'0x91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3'
dataSources:
  - kind: substrate/Runtime
    startBlock: 1
    mapping:
      file: ./dist/index.js
      handlers:
        - handler: handleEvent
          kind: substrate/EventHandler
          filter:
            module: balances
            method: Deposit
```

NB: Comment out genesisHash by prefixing with #. This is not required for now.

Step 4: Update the mappings file

The initialisation command pre-creates a sample mappings file with 3 functions, handleBlock, handleEvent and handleCall. Again, as we are only focusing on handleEvent, let’s delete the remaining functions.

We also need to make a few other changes. Because the Account entity (formally called the StarterEntity), was instantiated in the handleBlock function and we no longer have this, we need to instantiate this within our handleEvent function. We also need to update the argument we pass to the constructor.

```
let record = new
Account(event.extrinsic.block.block.header.hash.toString());
```

The mappingHandler.ts file should look like this:

```
import {SubstrateEvent} from "@subql/types";
import {Account} from "../types";
import {Balance} from "@polkadot/types/interfaces";

export async function handleEvent(event: SubstrateEvent): Promise<void>
{
    const {event: {data: [account, balance]}} = event;
     //Create a new Account entity with ID using block hash
    let record = new
Account(event.extrinsic.block.block.header.hash.toString());
    // Assign the Polkadot address to the account field
    record.account = account.toString();
    // Assign the balance to the balance field "type cast as Balance"
    record.balance = (balance as Balance).toBigInt();
    await record.save();
}
```

Step 5: Install the dependencies

Install the node dependencies by running the following commands:

```
yarn install
```

OR

```
npm install
```

Step 6: Generate the associated typescript

Next, we will generate the associated typescript with the following command:

```
yarn codegen
```

OR

```
npm run-script codegen
```

Step 7: Build the project

The next step is to build the project with the following command:

```
yarn build
```

OR

```
npm run-script build
```

This bundles the app into static files for production.

Step 8: Start the Docker container

Run the docker command to pull the images and to start the container.

```
yarn start:docker
```

Step 9: Run a query

Once the docker container is up and running, which could take a few minutes, open up your browser and navigate to www.localhost:3000.

This will open up a “playground” where you can create your query. Copy the example below.

```
query {
   accounts(first:10 orderBy:BALANCE_DESC){
    nodes{
      account
      balance
    }
  }
}
```

This should return something similar to the following:

```
{
  "data": {
    "accounts": {
      "nodes": [
        {
          "account": "13wY4rD88C3Xzd4brFMPkAMEMC3dSuAR2NC6PZ5BEsZ5t6rJ",
          "balance": "162804160"
        },
        {
          "account": "146YJHyD5cjFN77HrfKhxUFbU8WjApwk9ncGD6NbxE66vhMS",
          "balance": "130775360"
        },
        {
          "account": "146YJHyD5cjFN77HrfKhxUFbU8WjApwk9ncGD6NbxE66vhMS",
          "balance": "130644160"
        },
        {
          "account": "146YJHyD5cjFN77HrfKhxUFbU8WjApwk9ncGD6NbxE66vhMS",
          "balance": "117559360"
        },
        {
          "account": "12H7nsDUrJUSCQQJrTKAFfyCWSactiSdjoVUixqcd9CZHTGt",
          "balance": "117359360"
        },
        {
          "account": "146YJHyD5cjFN77HrfKhxUFbU8WjApwk9ncGD6NbxE66vhMS",
          "balance": "108648000"
        },
        {
          "account": "13wY4rD88C3Xzd4brFMPkAMEMC3dSuAR2NC6PZ5BEsZ5t6rJ",
          "balance": "108648000"
        },
        {
          "account": "12zSBXtK9evQRCG9Gsdr72RbqNzbNn2Suox2cTfugCLmWjqG",
          "balance": "108648000"
        },
        {
          "account": "15zF7zvdUiy2eYCgN6KWbv2SJPdbSP6vdHs1YTZDGjRcSMHN",
          "balance": "108448000"
        },
        {
          "account": "15zF7zvdUiy2eYCgN6KWbv2SJPdbSP6vdHs1YTZDGjRcSMHN",
          "balance": "108448000"
        }
      ]
    }
  }
}
```

If you have nothing returned, wait a few minutes for your node to index a few blocks.

What we have done here is queried for the balance of DOT tokens for all addresses (accounts) on the Polkadot mainnet blockchain. We have limited this to the first 10 and sorted it by the “richest” account holders first.

Bonus

As a bonus, try to aggregate the balances across addresses so you can find the total balance of an address.