



**Making the world's  
decentralised data more  
accessible.**

---

Lab Exercise Guide

## Table of Contents

Introduction	2
Pre-requisites	2
Exercise 1: Account Transfers (1-to-many)	2
High level steps	2
Detailed steps	2
Step 1: Initialize your project	2
Step 2: Update the graphql schema	3
Step 3: Update the manifest file (aka project.yaml)	3
Step 4: Update the mappings file	4
Step 5: Install the dependencies	7
Step 6: Generate the associated typescript	7
Step 7: Build the project	7
Step 8: Start the Docker container	7
Step 9: Run a query	7

## Introduction

In these exercises, we will take the starter project and focus on understanding a popular one to many entity relationships. We will create a project that allows us to query for accounts and determine how much was transferred to what receiving address.

## Pre-requisites

Completion of Module 2

## Exercise 1: Account Transfers (1-to-many)

### 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 transfers in the playground

### Detailed steps

#### Step 1: Initialize 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-transfers
? 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-transfers is ready
```

## Step 2: Update the graphql schema

Create an entity called “Account”. This account will contain multiple transfers. An account can be thought of as a Polkadot address owned by someone.

Transfers can be thought of as a transaction with an amount, a sender and a receiver. (Let’s ignore the sender for now). Here, we will obtain the amount transferred, the blockNumber and who it was sent to, which is also known as the receiver. The schema file should look like this:

```
type Account @entity {
  id: ID! #this primary key is set as the toAddress
}

type Transfer @entity {
  id: ID!
  amount: BigInt
  blockNumber: BigInt
  to: Account! # receiving address
}
```

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

Update the manifest file to only include the handleEvent handler and update the filter method to Transfer. This is because we only want to work with balance transfer events which will contain the data for transactions being transferred from one account to another.

```
specVersion: 1.0.0
name: account-transfers
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: Transfer
```

Note the inclusion of a dictionary and the exclusion of the genesisHash

#### Step 4: Update the mappings file

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

We also need to make a few other changes. Firstly, we need to understand that the balance.transfer event provides access to an array of data in the following format: [from, to, value]. This means we can access the values as follows:

```
const fromAddress = event.event.data[0];
const toAddress = event.event.data[1];
const amount = event.event.data[2];
```

Next, 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. However, we need to first test to see if this value is already in our database. This is because an event can contain multiple transfers to the SAME toAddress.

So we get the toAddress and if it does not exist, we save it to the database.

```
const toAccount = await Account.get(toAddress.toString());
if (!toAccount) {
  await new Account(toAddress.toString()).save();
}
```

Example of the account table in Postgres:

	id [PK] text
1	1121FVJsaJ9aD7oBL81dvRv2cjKQk4HMMTGjMQjyNhRRv8sW
2	11255FGt5mTokyoWwpuCE4Gwto5h1LM5AcBmKoc7GPkdSJ1
3	112E6RsivhtjPZeYetUsWUdWFWUorQQiWoPup48yaoby4hnk
4	112GEMoCceP4wbbrzCkx1vqiqjDyzYENE2r3erxz9qcEFzti

For the Transfer entity object, we set the primary key as the blocknumber+event.idx (which guarantees uniqueness) and then set the other fields of the Transfer entity object accordingly.

```
const transfer = new
Transfer(`${event.block.block.header.number.toNumber()}-${event.idx}`,
);
transfer.blockNumber = event.block.block.header.number.toBigInt();
transfer.toId = toAddress.toString();
transfer.amount = (amount as Balance).toBigInt();
await transfer.save();
```

Example of transfer table in Postgres:

	id [PK] text	amount numeric	block_number numeric	to_id text
1	7280565-2	400009691000	7280565	15kUt2i86LHRWckE3D9Bg1HZAoc2smhn1fwPzDERTb1BXAkX
2	7280566-2	23174700000000	7280566	14uh77yjhC3TLAE6KaCLvkjN7yFeUkejm7o7fdaSsggwD1ua
3	7280567-2	34192690000000	7280567	12sj9HTNQ7aiQoRg5wLyuemgvmFcrWeUJRI3aEUUnJLmAE56Y
4	7280568-2	400000000000	7280568	15kUt2i86LHRWckE3D9Bg1HZAoc2smhn1fwPzDERTb1BXAkX

The mappingHandler.ts file should look like this:

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

export async function handleEvent(event: SubstrateEvent): Promise<void>
{
  {
    // The balances.transfer event has the following payload \[from,
    to, value\] that we can access

    // const fromAddress = event.event.data[0];
    const toAddress = event.event.data[1];
    const amount = event.event.data[2];

    // query for toAddress from DB
    const toAccount = await Account.get(toAddress.toString());
    // if not in DB, instantiate a new Account object using the
    toAddress as a unique ID
    if (!toAccount) {
      await new Account(toAddress.toString()).save();
    }

    // instantiate a new Transfer object using the block number and
    event.idx as a unique ID
    const transfer = new
    Transfer(`${event.block.block.header.number.toNumber()}-${event.idx}`,
    );
    transfer.blockNumber =
    event.block.block.header.number.toBigInt();
    transfer.toId = toAddress.toString();
    transfer.amount = (amount as Balance).toBigInt();
    await transfer.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](http://www.localhost:3000).



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

```
query{
  accounts(first: 3){
    nodes{
      id
    }
  }
}
```

This will query the account entity returning the id. We have defined the id here as the “toAddress”, otherwise known as the receiving address. This will return something similar to the following:

```
{
  "data": {
    "accounts": {
      "nodes": [
        {
          "id": "11k5Gkwb9npuqWRq5Pyk51RSnRyskPrPtsyoCApteEUjNou"
        },
        {
          "id": "121dZJsfG7uNvszPSpYvBzwnrcF1P4ejjrE1G6FSWHqht5tC"
        },
        {
          "id": "121rwkQAH3yCD1EcaRgc3nELSoZn29RoTtCN55mcN7RkBA66"
        }
      ]
    }
  }
}
```

We can also query for all the transfers:

```
query{
  transfers(first: 3){
    nodes{
      id
      amount
      blockNumber
    }
  }
}
```

This will return something similar to the following:

```
{
  "data": {
    "transfers": {
      "nodes": [
        {
          "id": "7280565-2",
          "amount": "400009691000",
          "blockNumber": "7280565"
        },
        {
          "id": "7280566-2",
          "amount": "23174700000000",
          "blockNumber": "7280566"
        },
        {
          "id": "7280570-5",
          "amount": "400000000000",
          "blockNumber": "7280570"
        }
      ]
    }
  }
}
```

The magic lies in the ability to query the account id from within the transfer query. The example below shows that we are querying for transfers where we have an associated amount and blockNumber, but we can then link this to the receiving or “to” address as follows:














```
query{
  transfers(first: 3){
    nodes{
      id
      amount
      blockNumber
      to{
        id
      }
    }
  }
}
```

The query above returns the following results:

```
{
  "data": {
    "transfers": {
      "nodes": [
        {
          "id": "7280565-2",
          "amount": "400009691000",
          "blockNumber": "7280565",
          "to": {
            "id": "15kUt2i86LHRWCKE3D9Bg1HZAoc2smhn1fwPzDERTb1BXAkX"
          }
        },
        {
          "id": "7280566-2",
          "amount": "23174700000000",
          "blockNumber": "7280566",
          "to": {
            "id": "14uh77yjhC3TLAE6KaCLvkjN7yFeUkejm7o7fdaSsggWD1ua"
          }
        },
        {
          "id": "7280567-2",
          "amount": "3419269000000",
          "blockNumber": "7280567",
          "to": {
```

```
    "id": "12sj9HTNQ7aiQoRg5wLyuemgvmFcrWeUJRi3aEUnJLmAE56Y"  
  }  
}  
]  
}  
}  
}
```

Looking at the database schema also helps us understand what is happening. The accounts table is a standalone table containing just receiving addresses (accounts.id). The transfer table contains “to\_id” which links or points back to accounts.

- ▼  accounts
  - ▼  Columns (3)
    -  id
    -  created\_at
    -  updated\_at
  
- ▼  transfers
  - ▼  Columns (6)
    -  id
    -  amount
    -  block\_number
    -  to\_id
    -  created\_at
    -  updated\_at

In other words, one account links to many transfers or more verbosely stated, each unique Polkadot address that is stored in accounts.id links to one or more than one Polkadot address that has an associated amount and block number.