Developer Contest 1:

TON Block Explorer

Documentation

Introduction

This is a three part documentation aimed at both the contest jurors and possibly future developers that may want to expand on what we have build for this contest.

The first part will explain how to deploy the project in a local development environment and how to build it for production.

Secondly we will focus on the contest specifications and evaluate to what extent we completed them and explain some of our decisions during development

Lastly we will take a look at the technical aspects of the project as a developer reference.

Deploying the Project

Since you have found this documentation we assume you have already cloned the git repo onto your local machine. If you have not, the repository is located at: https://github.com/tondevs/TON-Blockchain-Explorer

To develop or build this project, **yarn** is required. You can install yarn with the following command: `npm install -g yarn`.

The following guide can also be found in the README.md within the repository.

Development

The TON Blockchain Explorer was developed using NuxtJS - https://nuxtjs.org/ To set up the project for development you can clone this repository using git clone <repo-url> and navigate your terminal into the project directory. Then run the following commands:

```
# Install all required dependecies
$ yarn

# Run the project locally on your system
$ yarn dev
```

Now you can visit `http://localhost:3000` to view the project and develop with hot reload and instant style changes. For more info on how to develop using NuxtJS visit: https://nuxtjs.org/docs/2.x/get-started/installation

Production

To get this project up and running on your production server you can use one of these two methods:

Method 1:

```
# Make sure you have yarn installed on your server. You can check it by
using
$ yarn -v

# Clone your repo on your server using git
$ git clone <repo-url>
$ cd <repo-name>

# Install all required dependecies
$ yarn

# Build the App using
$ yarn generate
```

If there are no errors generating the app you should see a dist folder inside your project root directory. Now you can point your web-server to the generated dist folder using **apache** or the following nginx config:

```
server {
  listen 80;
  location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
    try_files $uri $uri/ /index.html =404;
  }
}
```

Method 2:

```
# Install all required dependecies
$ yarn

# Build the App using
$ yarn generate
```

If there are no errors generating the app you should see a `dist` folder inside your project root directory. Now you can copy the content of the generated `dist` folder to your web-server and use **apache** or the following nginx config to point into the directory you uploaded the files to:

```
server {
  listen 80;
  location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
    try_files $uri $uri/ /index.html =404;
  }
}
```

Contest Specifications

1. *Observer for:*
    a. *Blocks - Completed*
    b. *Transactions - Completed*
    c. *Messages - Completed*
    d. *Account (Input and output transactions, sums, commissions) - Completed*
    e. *Validators - Completed*

All observer functionalities have been implemented.

2. *Link between related objects - Completed*

Each table or list includes links to related entities in the explorer.

3. Searching by any id (block, transaction, message, account) - Completed

The Home page features a search field that automatically determines which kind of id (block, transaction, message, account) has been input.

4. Export to .csv for any tables - Completed

Tables include and Export button at their top to export the currently displayed table.

5. Validator profitablity calculations per cycle, and view a history of such cycles

This requirement has been skipped due to time constraints.

6. Validator ranking (based on metrics like uptime and others included in the slashing conditions) - Completed

The current validators are ranked by how many blocks they have signed in relation to their uptime.

7. Clicking a validator wallet should show the previous stats for validation (including profitability and rewards for all cycle participated, accumulated tokens/rewards, lost/unstaked – returned due to maximum limit - tokens) - Partially completed

A validator's detailed view contains all values except profitability, cycle rewards and returned due to maximum limit.

8. Staking profitability dashboard for a user who specifies stake starting date, node, and number of tokens. Should include compounding. Weekly, monthly, yearly and future profitability projection also a bonus. - Partially Completed

The necessary UI has been built. Due to time constraints we were unable to finish and test the calculation.

9. Telegram and email login feature to remember the settings above. - Partially completed

We have consciously decided against a telegram or email login due to the following reasons:

- It would require a centralized database
- Setting up a database would complicate the project setup to an extent that is not justified by the provided functionality
- The login feature would only be used to save a few user inputs, which can just as easily be done with a Cookie

10. Automatic system for changing validator nickname - Completed

Using a hashing algorithm with the validator's public key as a seed we are converting the hashes to a human readable form.

11. Other important stats - Completed

We tried to find a balance between not overlading the UI, while still displaying all necessary and relevant data.

Summary

Since we started the contest late our goal was to make the codebase as accessible as possible to ensure other developers will be easily able to expand on what we have built, even though it meant trimming the specifications a little bit.
In our opinion we managed to create a welcoming project that can both stand on it's own and offers a great jumping off point to other developers.

Code Documentation

The code was entirely written without the TON SDK, but it can be easily integrated in the future if need be.

Frontend

The entire frontend is built with VueJS, which makes it very easy to expand or reuse.

If you are new to Vue you can refer to their official documentation at https://vuejs.org/v2/guide/

Routing System

Routes are automatically generated by nuxt. If you are looking for a sub-page, you will find it in the pages directory of the source code.

Example:
The two routes `BASE_URL + /leaderboard` and `BASE_URL + /validator/<id>` are generated by the following file structure:

- /
    - pages
        - leaderboard
            - index.vue
        - validator
            - _id
                - index.vue

For an in-depth look at Nuxt's routing system, please refer to https://nuxtjs.org/.

TONController

You will quickly notice that each page imports the TON Controller from `/assets/js/TONController.js` at the top of the script tag.
This is the source of all data we query and process across all sub pages.
It includes all the queries made to the subgraph and some helper functions to process the fetched data or refactor it.

Most of the functions simply take in a few query parameters for filters, consist of a query and a return statement for the fetched data.
The following functions fetches the last specified amount of messages on the network sent by a specified address since a certain point in time.

```
async getMessagesBy(base_url, account_addr, count = 10, fromTimeStamp =
undefined) {
    let messages;
    await axios.post(base_url, {
      query: `
      {
        messages(
          filter:{
            src:{eq:"${account_addr}"}
            ${fromTimeStamp ? 'created_at:{lt:' + fromTimeStamp + '}' : ''}
          }
          limit: ${count}
          orderBy: [{ path: "created_at", direction: DESC }]
        ) {
          id
          msg_type_name
          status_name
          value(format: DEC)
          src
          dst
          created_at
        }
      }
    `
    })
      .then((response) => {
        messages = response.data.data.messages;
      });
    return messages;
  }
```

Parameters:

- base_url: specifies which network to query
- account_addr: specifies the account that sent the messages
- count: how many messages to fetch
- fromTimestamp: fetch messages since this point

A response may look like this:

```
{
  "data": {
    "messages": [
      {
        "id":
"063a48aa6c5cbcd0509440e4b8c8eea6795ec3780a9cc3d6e3a292b012cb5924
",
        "msg_type_name": "ExtOut",
        "status_name": "Finalized",
        "value": null,
        "src":
"-1:ad47ea0c469262fe9cbdb07423dfd0e7173dd1dfd940eb2a962335ef85d2b
859",
        "dst": "",
        "created_at": 1604667602
      },
      ...
    ]
  }
}
```

Furthermore the controller includes helper functions to perform actions on the fetched data.

The following function will calculate the score for a specified validator object. This is used by the leaderboard page to display the Top 50 validators:

```
async calculateValidatorScore(baseUrl, validator, now)
  {
    const info = await this.getValidatorInfo(baseUrl,
validator.public_key);

    const since = info.utime_since;
    const uptime = now - since;

    const signaturesCount = info.aggregate_signed_masterchain_blocks;

    const score = (signaturesCount / uptime * 1000000);

    return {"node_id": info.node_id_hex, "score":
Math.ceil(score.toFixed(2))};
  }
```

Parameters:

- baseUrl: specifies the network
- validator: an object that contains the validator’s public key
- now: as a timestamp to calculate the validator’s uptime

Each Vue component or page will fetch the data it needs in this fashion and process it according to it needs.