Brightsign REST-API Test Application

Build requirements

Implementation is made with C++ and tested with Linux/gcc.

Following libraries are required to be installed. (Tested on Linux)

- libcpr (REST-API management)
- nlohmann::json JSON-parsing

How to Build

```
- mkdir build
- cd build
- cmake ..
- make
```

Source Code Structure

Application is divided to 4 main parts in source code

1. JSON data parsing and calculation classes to provide responses for the requested questions.

   JSON parsing and calculation classes are located in separate bsdjsonparser sub-directory. They all inherit from the same base class as some of the calculations can use similar type of logic. These are build as a separately library so they could be used potentially also by another application.

2. REST-API management class

   RestAPIHandler.hpp and RestAPIHandler.cpp. In addition of implementing the GET-method required these can also be used in verbose logging mode to save the HTTP request and response to log-file if user request that via command line parameter.

3. Brightsign.cpp is the main class

   Parses the user-input and manages the main logic.

4. tests

   Test code that could be used to read the JSON data from the file for example instead of reading from the REST-API server. Usefull for checking error situations for example by modifying the JSON data from the files.

Application Example Usages

Requirements of the application specified that application must accept the endpoint to be queried as a command line parameter. This is implemented in a way that the endpoint url is expected to always be the last parameter.

In addition it is possible to specify also some other parameters that will be specified later for example to restrict what data is calculated and whether the data is tried to poll periodically or only once per

execution.

Get test data from REST-API and perform all 5 calculations.

Results of calculations are provided out as a single JSON Data structure which have following subnodes to answer for the requested questions

```
./brightsign http://test.brightsign.io:3000
```

- average_age_of_all_users_per_city
- average_number_of_friends_per_city
- user_with_the_most_friends_per_city
- most_common_first_name_in_all cities
- most_common_hobby_of_all_friends_of_users_in_all_cities

Show the help for command line parameters

```
./brightsign --help
Usage: brightsign [options] <connection url>
Options:
  --verbose,        Show verbose message and create log file
connection_data_<datetime>.log
  --poll <seconds>, Poll the data periodically. Specified poll interval must be a
positive number.
  --mask <bitmask>, Bitmask from 0 to 31 to specify which from 5 calculations to
performed for the retrieved data.
  --help            Show this help message
```

Execute only the last 2 calculations on every 5 seconds

```
./brightsign --mask 24 --poll 5 http://test.brightsign.io:3000

{"most_common_first_name_in_all cities":
["Mia"],"most_common_hobby_of_all_friends_of_users_in_all_cities":["Movie Watching"]}
{"most_common_first_name_in_all cities":
["Mia"],"most_common_hobby_of_all_friends_of_users_in_all_cities":["Movie Watching"]}
```

Generate verbose log from the request send to server and response received.

Log file is in format connection_data_20250224_192933.log

```
./brightsign --verbose 5 http://test.brightsign.io:3000
```

JSON Data format and REST-API Protocol

JSON data format seemed to be pretty simple for parsing data, biggest problem was that the amount of data in single JSON read was quite big making the reading in that way harder for simple tests to verify it. The structure of JSON data is following.

In node-level the data is divided to 3 bigger sub-nodes:

- Users

- Friends
- Friends Hobbies

User's are uniquely idendified from each others by their ID, as many users can have same name. Users or friends name can not be itself be used to identify each others reliably as there can be multiple users and friends with the same name. Friends structure does also not contain the user-id, so friends referred does not neccessarily need to be themself listed as a User. In addition same User and Friend names can exist in multiple cities.

Each hobby name and city name are identified from each others also by letters, there is no any IDs used to verify them from each others. Therefore the hobby-names and city names must also match letter-by-letter to be considered to be same.

I did some small tests for trying to find alternative endpoints for reading a data for single user friends for example by testing things like /url/friends/?id=40001 but could not find an alternative method to get a data in smaller chunks.

Transmitted data seems to be quite big and that could itself cause some memory problems and delays when parsing it. It would propably be good to offer futher APIs to be able to read the data in smaller peaces for example by allowing to read for each city separately for example by using.

Security Considerations

First steps to increase the security would be:

- Allow only the connection over an end-to-end encrypted connection (HTTPS, for example)
- Add a server certificate so that clients can verify the server they communicate with
- Add authentication requirements, for example, by using OAUTH2
- Add support for some type of session token that needs to be transmitted on each request

Current Security Issues

The REST-API server is currently connecting to another REST-API server over a plain HTTP connection. This means that all data is transmitted in plain text format without any kind of encryption. As a result, the data can potentially be read and modified, for example, using a "man-in-the-middle" attack while requests and responses are transferred over the internet.

Another problem with using the plain HTTP protocol is that no server certificate check is implemented.

Authentication and Session Management

A third problem is the lack of authentication, such as OAUTH2, which ensures that only authorized users—those on the list of allowed users—can connect after authenticating themselves. Additionally, it would be beneficial to implement extra parameters, like session tokens, to verify that only authorized users can access data. Users should also be required to renew their credentials periodically. This would reduce the possibility of DDoS attacks, as unauthenticated users would not be able to fetch large amounts of JSON data.

REST-API and JSON Protocol Security

Another security concern is the implementation of the REST-API and JSON protocols themselves. As the example application is relatively simple and only uses the GET method, I considered implementing the REST client purely myself by using the sockets in the C++ library, to avoid requiring the user to install any dependencies.

However, as demo applications often evolve into product versions, I decided it is better to use a well-known library instead. These libraries hide many details, such as connection keep-alive, which might be needed in the future. Additionally, they are more widely researched and patched for security vulnerabilities. The popularity of these libraries may also increase the chances of potential security issues being found, but they are also more likely to be fixed quickly since many operating systems maintain these libraries for security updates.

Input Validation

Additionally, I performed some testing with broken user input parameters for the application. All user parameters are checked, and if, for example, there is an error parsing the expected integer, the application will catch and handle the error.

Local JSON Testing

I also saved some examples of the received JSON data to local files and implemented a test directory to allow loading JSON data from these local files instead of fetching it from the REST-API server. This setup allowed me to modify the data to check that errors in JSON parsing are handled properly—by catching the exception and printing an error message for the user.

Example usage:

```
./tests/local_json_file_tester ../tests/brightsign_example_invalid_age.json
```