HTTP BOARD
HTTP NÁSTENKA

AUTHOR MAROŠ ORSÁK
AUTOR PRÁCE

BRNO 2019

Contents

Chapter 1

Introduction

As the time pass industry progresses and creating a REST API becomes indispensable feature. The goal of this project is to create RESTful web service using CRUD methods. Application use HTTP protocol, which is build on the top of the previously mentioned REST API feature. Project is written in C++ programming language using version C++14 in Object Oriented Programming style.

Chapter 2

Design of the application

In this chapter we will take a look on design. My inspiration of creating overwhelming architecture was do it in web design way but before i started i have read a lot of Articles[4][3], RFC[5] and also gladly metioned two books [1][2]. My application was divided into a two logically separate modules and few units as follows:

- Manager of client
  - mainClient.cpp
  - ParseArgs.cpp /h
  - Module client
    - Client.cpp /.h
    - ClientController.cpp /.h
- Manager of server
  - mainServer.cpp
  - ParseArgs.cpp /h
  - Module server
    - Server.cpp/.h
    - Controller.cpp/.h
    - Boards.cpp/.h
    - Board.cpp/.h
    - Commend.cpp /.h

On the top of these modules we have two super classes which are managing them. Module client has class ‘mainClient.cpp‘ and similarly server has ‘mainServer.cpp‘. These two classes has shared class ‘ParseArgs.cpp/.h‘ which is as the name suggest checking input parameters provided by the user.

2.1 Brief description of classes

mainServer.cpp - used for managing whole server instance
mainClient.cpp - used for managing whole client instance
ParseArgs.cpp/.h - used for checking input parameters provided by the user.

Module server

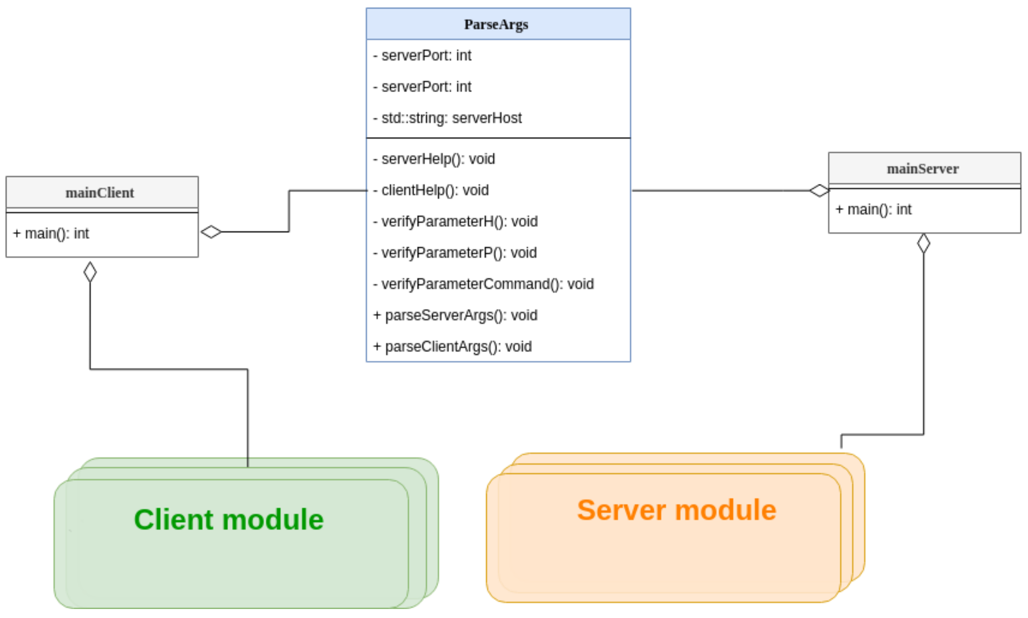
Server.cpp/.h - used for managing for accepting user communication, parsing request of the user provided by REST calls, storing data into data structures such as hash map and so on.
Controller.cpp/.h - used for handling all REST calls and dividing them into separated parts based on HTTP methods.
Boards.cpp/.h - used for managing all boards
Board.cpp/.h - used for managing board
Commend.cpp/.h - used for managing commend

Module client

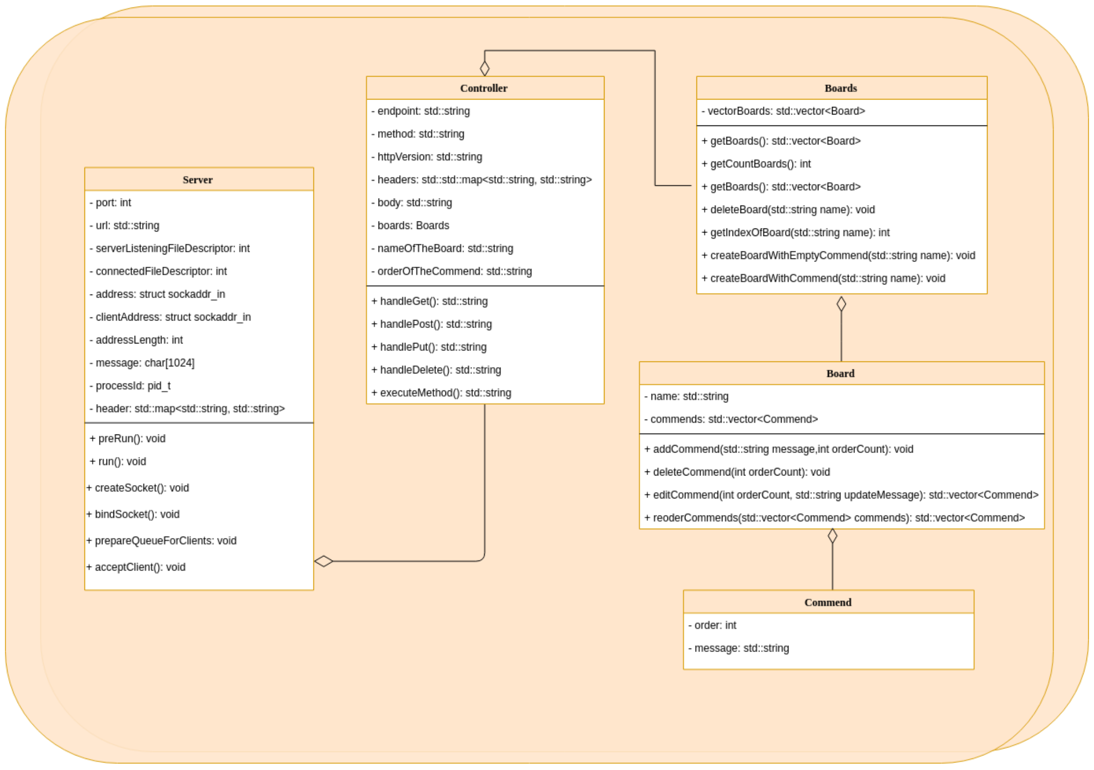
Client.cpp/.h - used for managing for communication and connecting to the server with specific properties provided by the user
ClientController.cpp/.h - used for building REST API requests to the server, based on user provided parameters in command line interface.

2.2 Class diagrams

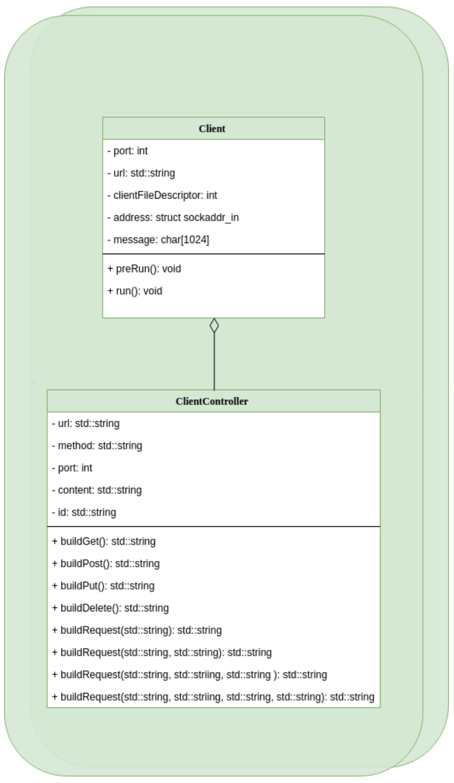
General Diagram


ParseArgs
- serverPort: int
- serverPort: int
- std::string: serverHost
- serverHelp(): void
- clientHelp(): void
- verifyParameterH(): void
- verifyParameterP(): void
- verifyParameterCommand(): void
+ parseServerArgs(): void
+ parseClientArgs(): void
mainClient
+ main(): int
mainServer
+ main(): int
Client module
Server module

Server diagram


Controller
- endpoint: std::string
- method: std::string
- httpVersion: std::string
- headers: std::std::map<std::string, std::string>
- body: std::string
- boards: Boards
- nameOfTheBoard: std::string
- orderOfTheCommend: std::string
+ handleGet(): std::string
+ handlePost(): std::string
+ handlePut(): std::string
+ handleDelete(): std::string
+ executeMethod(): std::string
Boards
- vectorBoards: std::vector<Board>
+ getBoards(): std::vector<Board>
+ getCountBoards(): int
+ getBoards(): std::vector<Board>
+ deleteBoard(std::string name): void
+ getIndexOfBoard(std::string name): int
+ createBoardWithEmptyCommend(std::string name): void
+ createBoardWithCommend(std::string name): void
Server
- port: int
- url: std::string
- serverListeningFileDescriptor: int
- connectedFileDescriptor: int
- address: struct sockaddr_in
- clientAddress: struct sockaddr_in
- addressLength: int
- message: char[1024]
- processId: pid_t
- header: std::map<std::string, std::string>
+ preRun(): void
+ run(): void
+ createSocket(): void
+ bindSocket(): void
+ prepareQueueForClients: void
+ acceptClient(): void
Board
- name: std::string
- commends: std::vector<Commend>
+ addCommend(std::string message,int orderCount): void
+ deleteCommend(int orderCount): void
+ editCommend(int orderCount, std::string updateMessage): std::vector<Commend>
+ reoderCommends(std::vector<Commend> commends): std::vector<Commend>
Commend
- order: int
- message: std::string

Client diagram


Client
- port: int
- url: std::string
- clientFileDescriptor: int
- address: struct sockaddr_in
- message: char[1024]
+ preRun(): void
+ run(): void
ClientController
- url: std::string
- method: std::string
- port: int
- content: std::string
- id: std::string
+ buildGet(): std::string
+ buildPost(): std::string
+ buildPut(): std::string
+ buildDelete(): std::string
+ buildRequest(std::string): std::string
+ buildRequest(std::string, std::string): std::string
+ buildRequest(std::string, std::striing, std::string): std::string
+ buildRequest(std::string, std::striing, std::string, std::string): std::string

Chapter 3

Implementation

In this section we will take a look in detail on architecture and also i will describe interesting parts of my work on this application. Firstly let me start with the detailed description of architecture.

The whole concept was to create application based on Model, View, Controller design pattern but eventually i realised that View and Model components would be really useful for that use case. I decided to create base class Controller for each module. The main usage of this class from the servers side is to handle every request from the client. Moreover this also means, that controller dividing the enpoints in the methods for CRUD in other words are basic methods of persistent storage.

3.1 Server

Everything starts in the mainServer.cpp where we instantiate two objects. First one is ParseArgs which is object for parsing input from the user. Based on count of parameters we are dividing the logic. In ParseArgs.cpp is interface which is defining methods for example verifyParameterH, verifyParameterP and lastly verifyParameterCommand as the name says verifying each parameter. If the client specify invalid input parameters the server commmand line interface will response with help.

Afterwards in mainServer.cpp we are creating instance of Server. Futhermore in the next lines we set port number of serve, execute method preRun which basically initialize some fields of the class and also execute the most significant method run. In this method we create file descriptor(socket), bind the socket and also create queue for coming clients. Server is running in the neverending loop where he constantly parse the request from client put each header to the map do some operation such as divide method, decide which endpoint will go to some specific line of code and eventually build response. The response are building inside Controller class which i described early. I have created three helpful classes which will represent all logic and each class will have some responsibility. Boards class is taking care of all boards and has interface such as delete specfic board, creating board, checking if the board is present and so on. On the other hand the Board class is responsible for adding commend to the concrete board, deleting commend, reordering commends and finally also edit the commend.

Defined application programming interface

- GET /boards - show all boards
- POST /boards/<name> - create new empty board with the name <name>
- DELETE /boards/<name> - delete board name with the <name> and erase whole content
- GET /board/<name> - show content board with name <name>
- POST /board/<name> - create commend which is inserted to the end of the board
- DELETE /board/name/<id> - deletes content of commend with <order/id> in board <name>
- PUT /board/<name>/<id> - update commend in board with name <name> with <order/id> and replace it with content <content>

Supporting status codes

- Status 200 OK - response code which means that everything was fine and operation succeed
  - In scope of this application you can get 200 code in following cases:
    - /GET boards
    - POST /boards/<name>
    - DELETE /boards/<name>
    - GET /board/<name>
    - POST /board/<name>
    - DELETE /board/name/<id>
    - PUT /board/<name>/<id>
- Status 201 CREATED - response code which means that everything was fine and resource was created
  - In scope of this application you can get 201 code in following cases:
    - POST /boards/<name>
    - POST /board/<name>
- Status 400 BAD REQUEST - response code which means that client send invalid request to the server for example malformed request syntax. This Status code is not supported, because i am handling this type of error before client will started communication with the server. On the other hand if we communicate with server using web browser or curl utility it's possible. More in subsection 4.4.2 Test by curl
- Status 404 NOT FOUND - response code which means that server can not find requested resource on the server
  - In scope of this application you can get 404 code in following cases:
    - DELETE /boards/<name>

        * GET /board/<name>
        * DELETE /board/name/<id>
        * PUT /board/<name>/<id>

- Status 409 CONFLICT - response code which means that server has already this type of resource with the specific name
    - In scope of this application you can get 201 code in following cases:
        * POST /board/<name>

3.2 Client

The same design as was describe in section Server is applied to the client module. In mainClient.cpp we are parsing input from the client and then each of provided parameters we have to map it in CRUD methods. Essentially, what is different between server that we need only two classes for whole task. What is worth mentioning is that in run method in client i have defined one method for multiple implementations. This is called in Object Oriented Programming - Overloading.

Chapter 4

User manual

Application must be before we use it building with help of Makefile. Once build is succeed we can start server instance and after that we can use our client command line interface. I will continue to describe this approach step by step.

4.1 Before starting binaries

1. move to root directory of the project
2. execute command **make**

Now we are ready to use compiled binaries to start server and client.

4.2 Server

Server has to be start with mandatory parameters as follows:

- ./isaserver -p <port-number>
  - where <port-number> is number higher than 1024 and lower than 65536.

Print usage of server:

- ./isaserver -h

4.3 Client

Client has a little bit complicated CLI and more mandatory parameters as follows:

- ./isaclient -H <hostname/ip-address> -p <port-number> <command> where <command> could be as follows:
  1. boards —– LIST ALL BOARDS
  2. board add boardName —– ADD NEW BOARD
  3. board delete boardName —– DELETE BOARD
  4. board list boardName —– LIST COMMENDS OF BOARD

5. item add boardName content —– ADD COMMEND TO BOARD
6. item delete boardName id —– DELETE COMMEND OF BOARD
7. item update boardName id/order content —– UPDATE COMMEND OF BOARD

Print usage of client:

- ./isaclient -h

4.4 Examples

Everything you will find in README file with scenarios, which you can follow.

Chapter 5

Conclusion

The goal of this project was to create HTTP server and REST API client and from my point of view this goal i accomplished. Creating an architecture for this application was very easy when taking into account that I was doing an HTTP web server in the past using certain frameworks such as Quarkus, Vert.x, Spring and so on. This project gives me better understanding of how exactly HTTP works on the application layer.

Improvements

If this application will extend it would be great to consider add new component called Model in both sides to manage data processing. I decided to not do that yet because it would be memory consuming creating instance of Model because of just returning headers(strings). The application can grow and we can imagine if it will be some integration of database this model would be ideal decision.

Bibliography

[1] Brian Totty, M. S. A. A. S. R. *HTTP: The Definitive Guide.* O'Reilly Media, 2009. ISBN 1565925092.

[2] Matoušek, P. *Síťové aplikácie a jejich architektúra.* VUTIUM, 2014. ISBN 978-80-214-3766-1.

[3] RestApiTutorial.com. *Learn REST: A RESTful Tutorial.* [Online; visited 14.10.2019]. Available at: https://www.restapitutorial.com/.

[4] RestApiTutorial.com. *Rest api tutorial.* [Online; visited 12.10.2019]. Available at: https://restfulapi.net/http-methods/.

[5] Roy T. Fielding, J. C. M. H. F. N. L. M. P. J. L. T. B.-L. *Hypertext Transfer Protocol – HTTP/1.1.*