# Design Documentation

## Homedork - Interactive House

# Revision History

| Name | Associated Letter |
|---|---|
| Lukas Olsson | A |
| Wills Ekanem | B |
| Bujar Rabushaj | C |
| Besnik Rabushaj | D |

| Date | Version | Description | Author |
|---|---|---|---|
| 16/9/2021 | 0.0 | Initial Design Draft | A, B, C, D |
| 6/10/2021 | 0.1 | Secondary Revision | A, B, C, D |
| 21/10/2021 | 0.2 | Added API resources class diagrams and updated existing diagrams. Clarified API and DBS communication method. | B |
| | | | |

# Design item List

| Requirement Name | Priority |
|---|---|
| D1. System Architecture | Essential |
| D2. Server Architecture | Essential |
| D3. Communication Design | Essential |
| D4. Class Diagrams | Essential |

# Design Item Descriptions

## D1 System Architecture

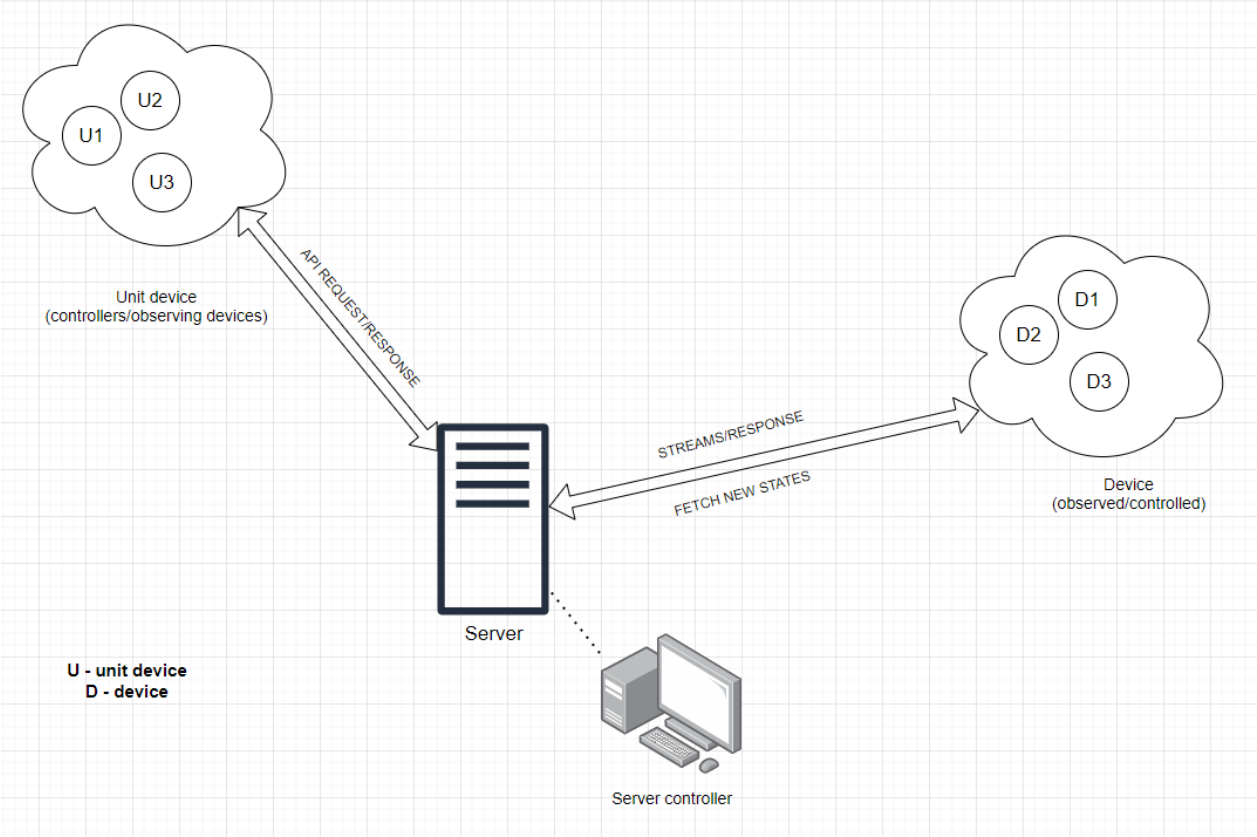


*Figure 1: System Architecture*

*Figure 1* shows a native/far view of the architecture of the entire project without going into the sub-components of the main components of the smart house system. API request/response are the standard communication format between and API server and client.

Using this structure will result on a turn of unnecessary load on the server which result in us breaking down into several other components which handle specific tasks in a chain like mechanism.

The server in figure 1 if deeped are two servers, an API server for handling http req/res and a database server which holds current/updated state of all connected devices and user related information.

## D2 Server Architecture

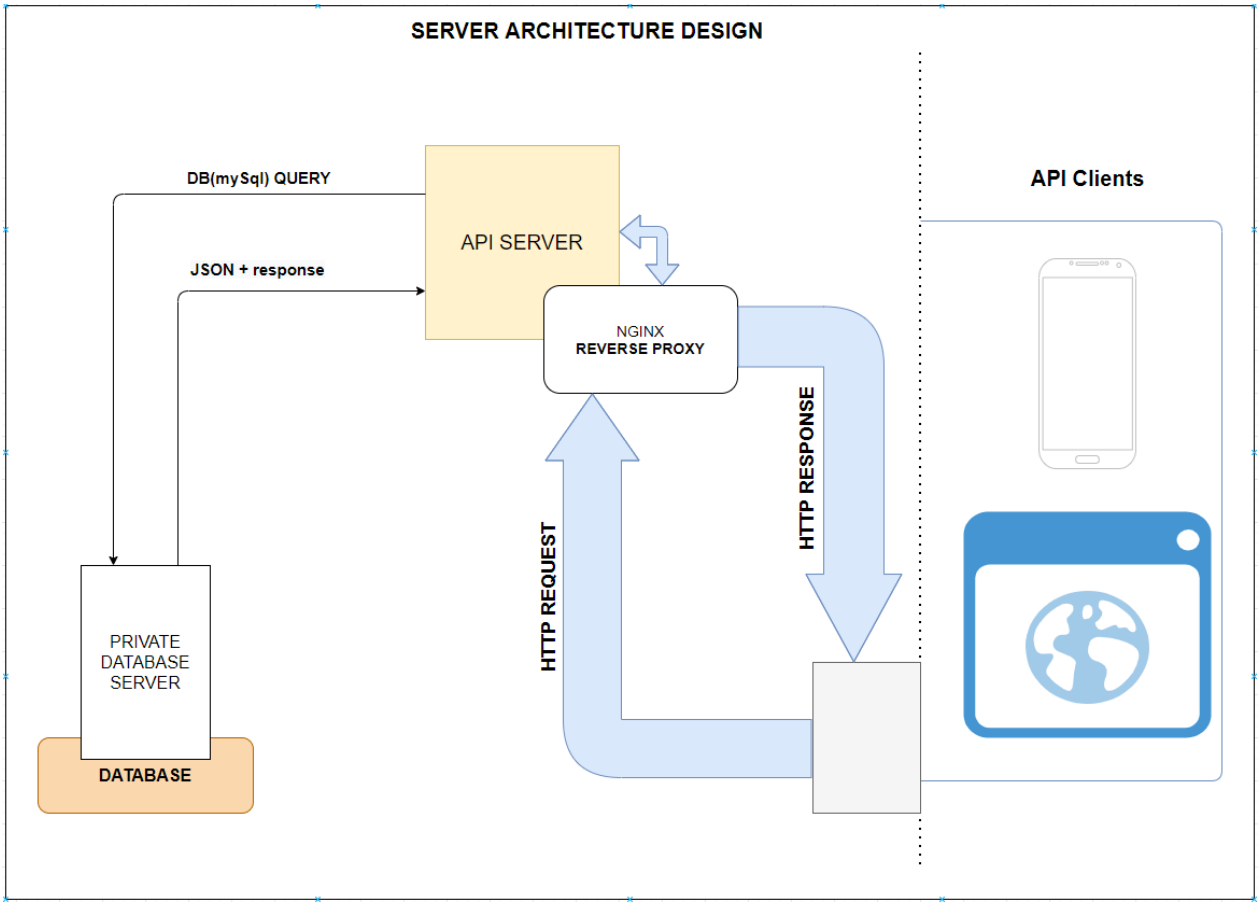


*Figure 2: Server Architecture*

An API server is deployed between the **private database server** running in the background to handle **API clients'** requests and response and the unit devices (Clients). The database in this case only has one function and that is to store/update states of devices, mostly **DB CRUD** operations corresponding to the HTTP request sent (GET, POST, PUT, DELETE). On request sent to API server from any client, a query string is built/sent to the DB server, DB server responds with a json object of the fetched object and a response string(code)[control message], then the API maps the json object to it appropriate representation and responds with json back to the client.

A NGINX server is also deployed right in front of the API, on the same machine and acts as a reverse proxy (on which the clients are connected to) to prevent the API from being accessible from the public internet directly. NGINX helps with load balancing in case of future expansion of the project.

## D3 Communication Design

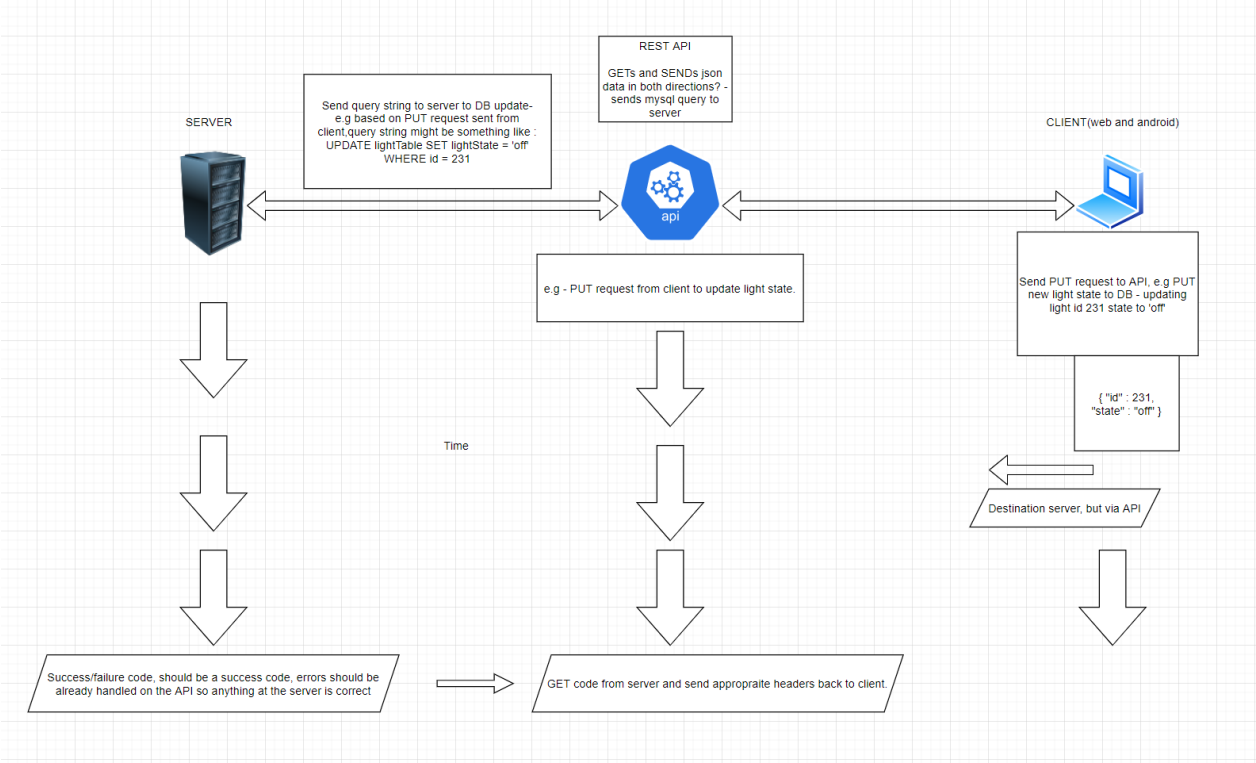


*Figure 3: Communication Design*

*Figure 3* is practically a scenario like diagram. User decides to turn lamp with "id=231" -"off".

A JSON object of this is sent up to the API with a HTTP put method since its a regular update, the API builds the appropriate query string to update the **state** of device with **id=231.**

The query is sent down to the DB server via an encrypted channel with already generated and pre-exchanged keys. Sever receives, handles decryption, and passes the received query string into a **statement(Java class)**, and processes the received data from the MySQL DB if any, wraps into a json object, prepends a **control message** understood at the API level, encrypt, and sends it to the API.

Example.

```
status code : 200 (OKAY)
{
    "id"     : "231",
    "state"  : "off",
    "level"  : "xxx",
    "userId" : "xxx-xxx"
}
```

*Figure 4 - Example of response from DB server to API*

| Status code | Message |
|---|---|
| 200 | okay |
| 300 | invalid ID |
| 350 | empty |
| 770 | operation not server supported |

*Figure 5 - Database Server control messages and codes*

API gets it, parses the json object to its correct POJO representation using GSON or some other lib and passes it between utility classes* and the corresponding service and resource class all the way down to the API client (Unit device).

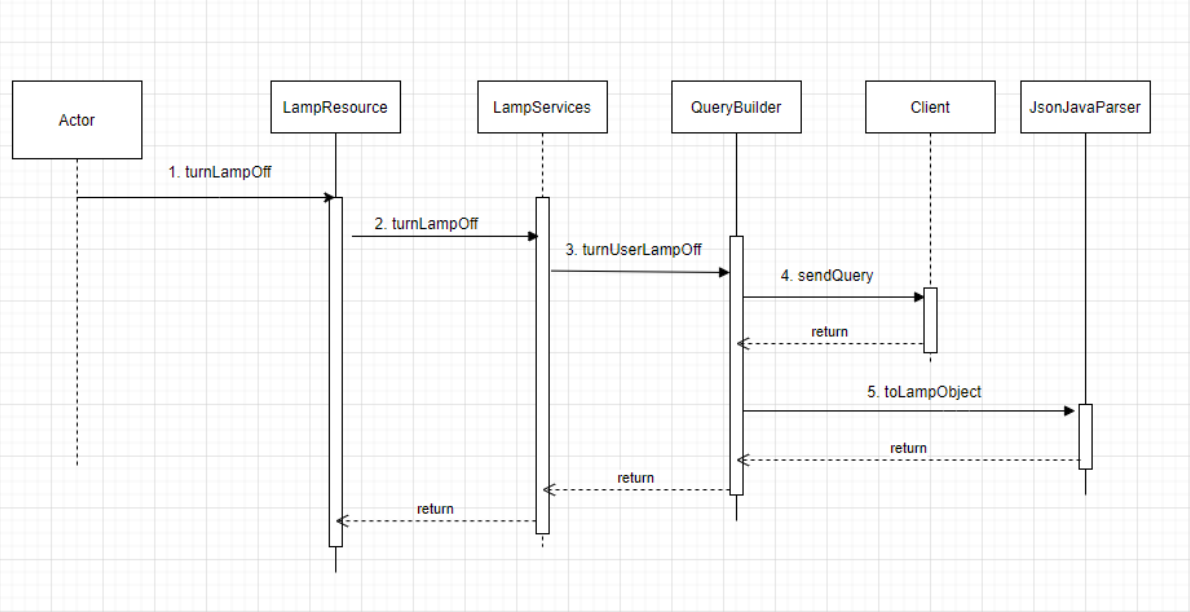


*Figure 6 - Sequence diagram to turn off user's lamp*

## API communication with Database server.

The API and the database server communicate and exchange data via an **encrypted socket channel**. The API acts as a client to the server socket running on the DB server and sends encrypted query strings that corresponds to the **http requests** sent down from a unit client. Private key encryption is used between the API and DB server, keys are rightfully exchanged before the system is up and running. **AES** encryption is used with padding. 256 bits key sizes are used for both encryption/decryption on both ends.

## D4 Class Diagrams

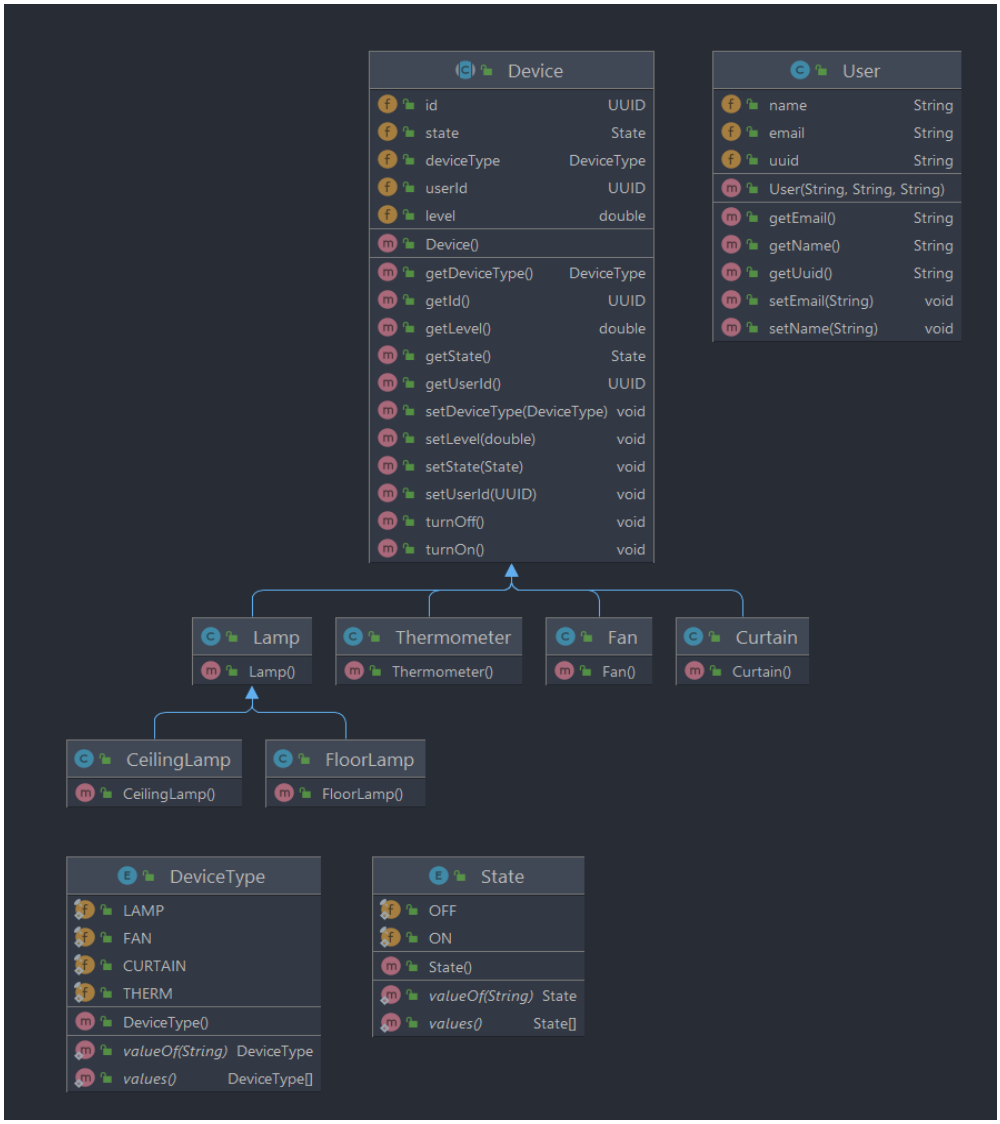


*Figure 7: Model class Diagrams*

*Figure 7* shows an the already implemented classes, more will come after we are sure all types of devices will be working with. An **abstract device class** is used to store all devices since all devices(fan, lamp) practically share common attributes like **state**, **level**, and **ids**. A device and be turned off/on directly via inherited methods. Device level e.g., lamp brightness levels can be set directly through its setter and gotten via its getter. An Enum is used to set the exact type of a device (**FAN**, **LAMP**,**THERM**,**CURTAIN** ...), attributes that are not common between different type of devices are set as nullable and the device type Enum is present so that only fields that are not null are retrieved when needed.

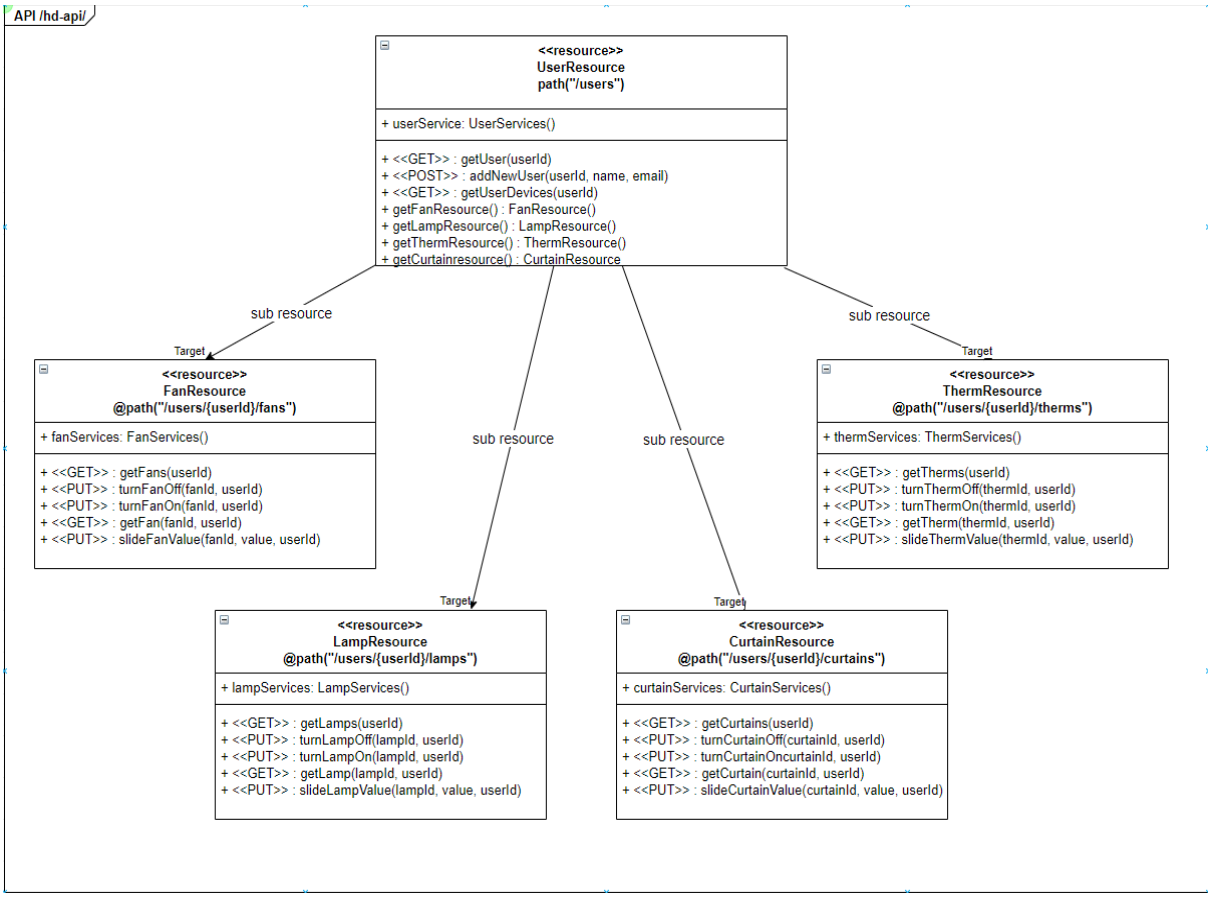


*Figure 8 - API resource package and classes*

Homedork has "**User Resource**" class as its root class, all users and devices related operations are firstly routed via this class since variable userId is used as a unique user identifier. User related operations like posting a new user, get user profile, get user devices, edit user information are handled in the **UserResource** and **UserServices** classes, then down to the **QueryBuilder** and **Client** classes for communication with the Database server.

Devices related like turn off/on a fan, lamp, curtain, or Music player are routed to its corresponding sub resource. A fan related operation like "turn a fan off" is routed to the **FanResource** class from the **root resource** class(**UserResource**).

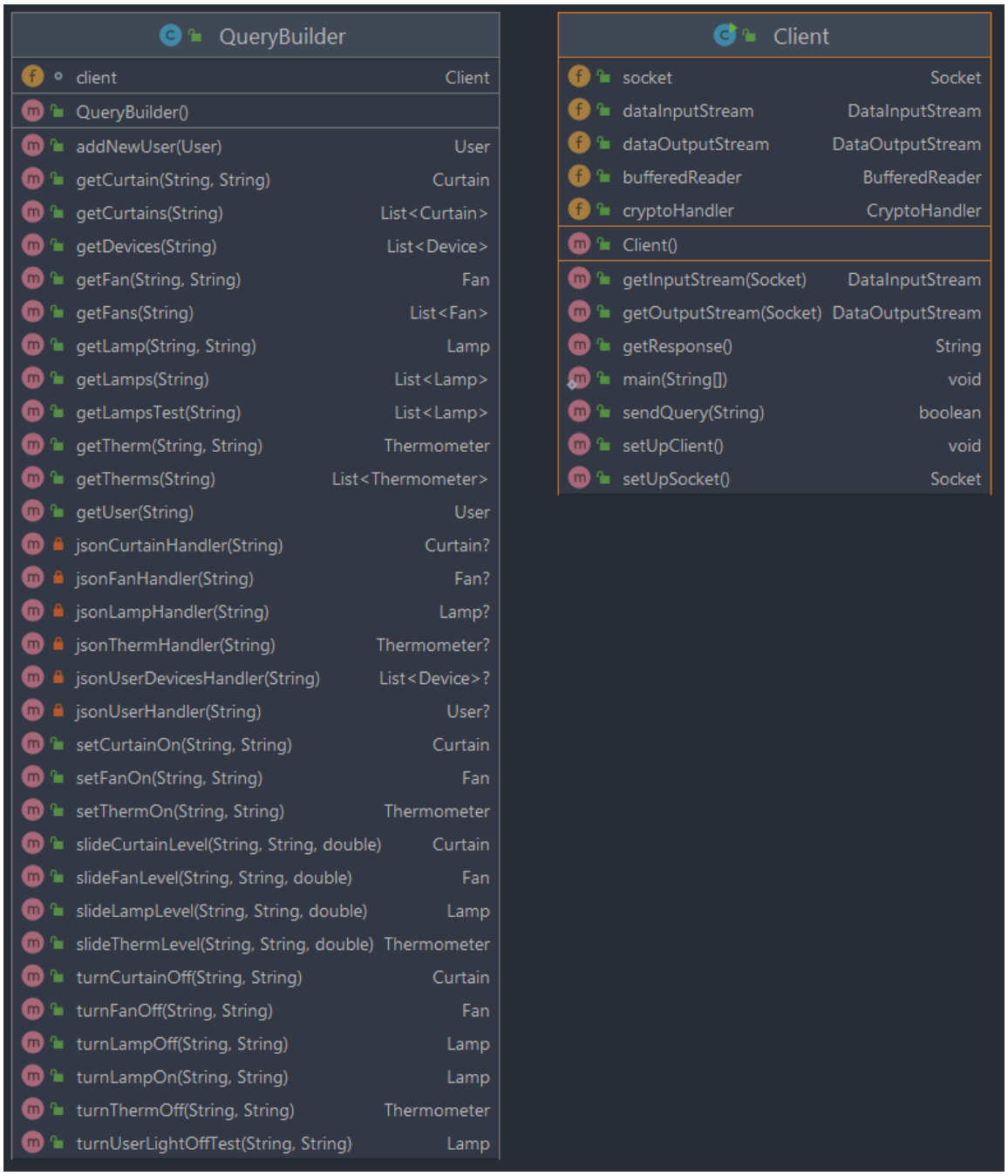


*Figure 9 - Client and Query Builder class diagrams*

*Figure 9* shows a query builder class that builds queries based on parameters to be sent to the DB server side for database operations and gets the response afterwards via the client class **getResponse() – returns json object** method. It is then parsed by a custom "**JSONJavaParser**" class using the GSON lib to its appropriate java object.

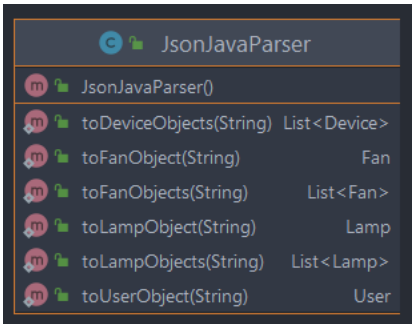


*Figure 10 - JSONJavaParser class diagram*

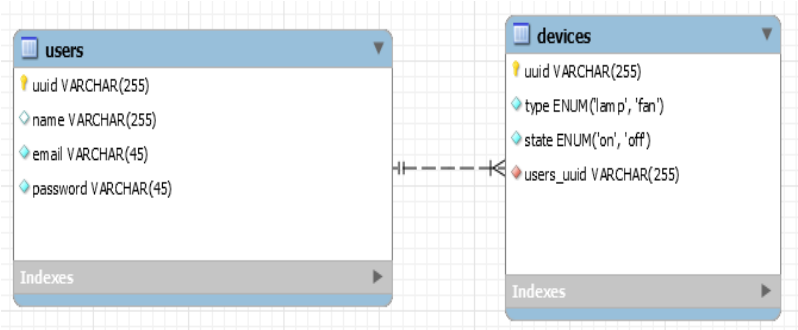


*Figure 11 - Database tables*

*Figure 10* is the **JsonJavaParser** class that parses json object responses received from the database server into java objects.

*Figure 11* shows the tables we have in our already made MySQL database.