

Design Documentation – Server/database group

Homedork – Interactive Smart 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	1.0	Initial Design Draft	A, B, C, D
06/10/2021	1.1	Secondary Revision	A, B, C, D
21/10/2021	1.2	Added API resources class diagrams and updated existing diagrams. Clarified API and DBS communication method.	B
12/11/2021	1.3	Design updates. <ul style="list-style-type: none">- Server arch: addition of hub client for communication with local hub.- General communication design update.- Server response to device update.- Device class update.- Added communication flow chart	B
7/12/2021	1.4	Updates. <ul style="list-style-type: none">- Diagrams update.- Sequence diagrams.	B
10/01/2022	1.5	Updates. -Query Builder Class Update.	A

Design item List

Requirement Name	Priority
D1. System Architecture	Essential
D2. Server Architecture	Essential
D3. Communication Design	Essential
D4. Sequence Diagrams	Essential
D5. Class Diagrams	Essential

Design Item Descriptions

D1 System Architecture

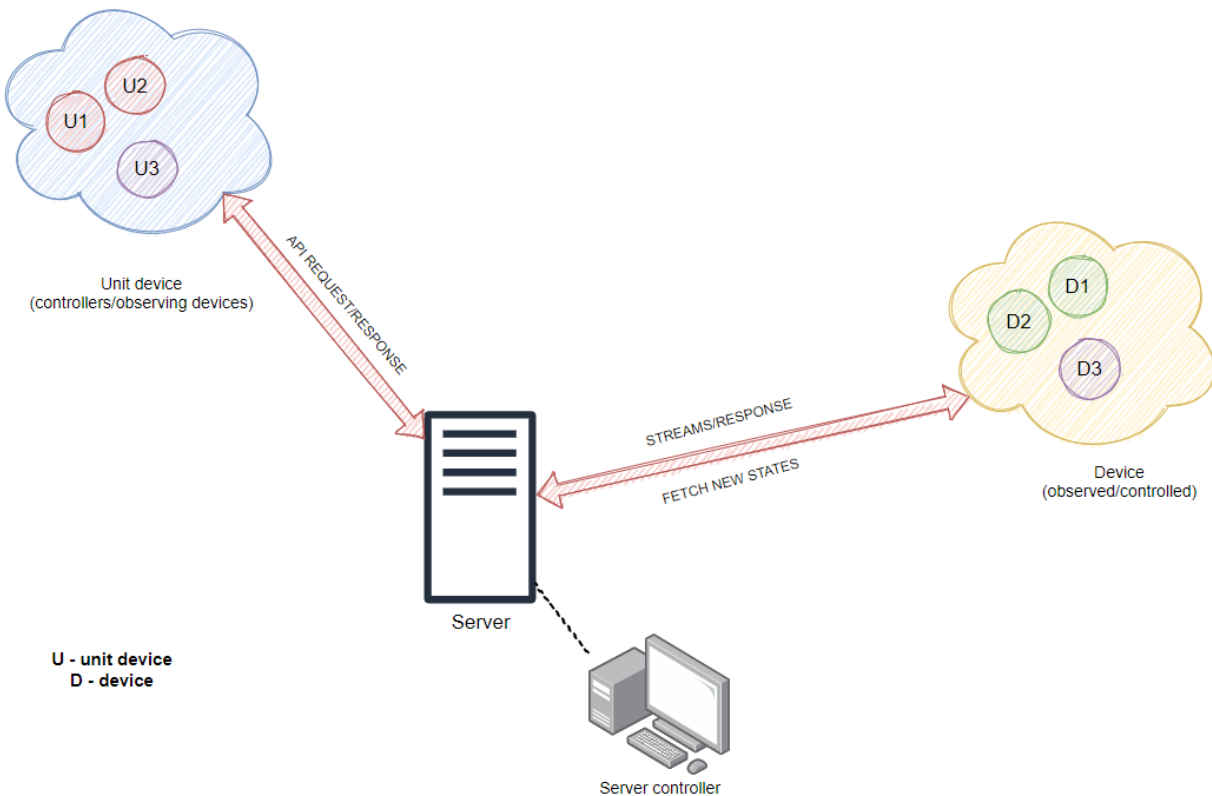


Figure 1: System Architecture

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

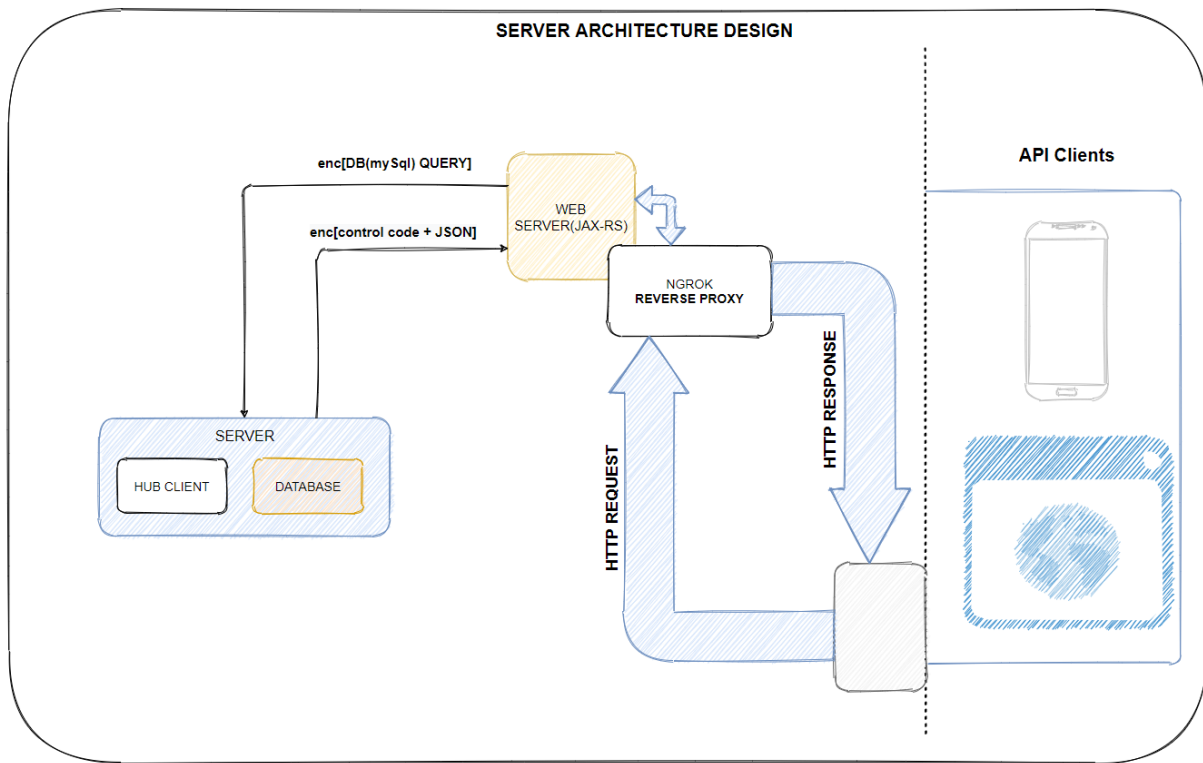


Figure 2: Server Architecture

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. NGROK helps with load balancing in case of future expansion of the project.

D3 Communication Design

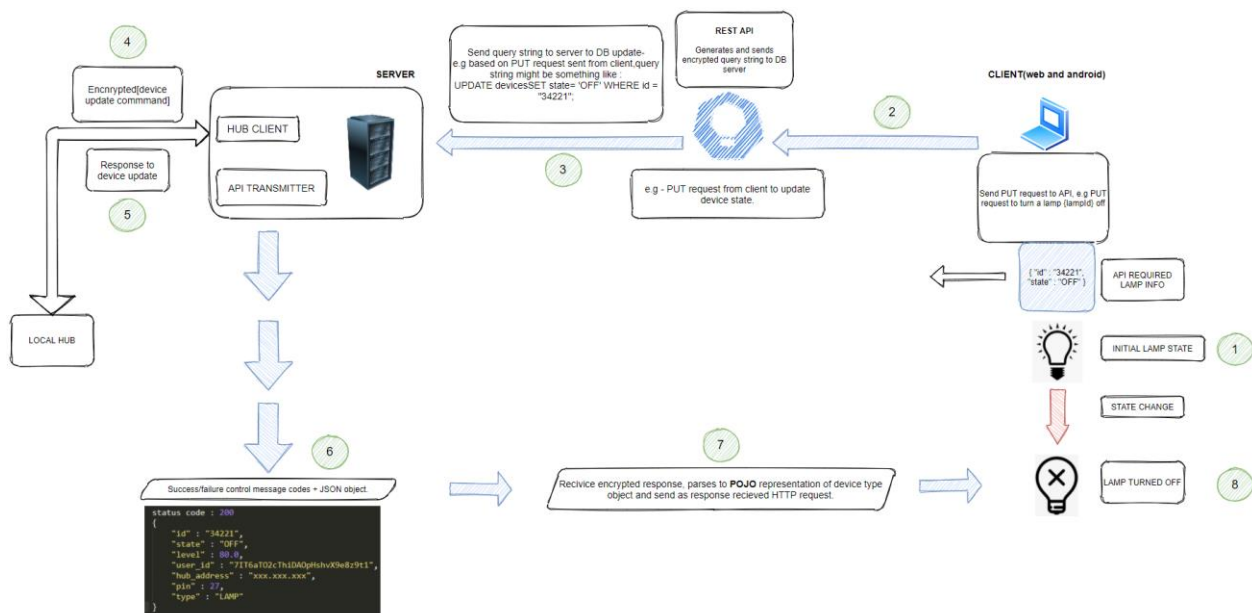


Figure 3: Communication Design

Figure 3 is practically a scenario like diagram. User decides to turn lamp with “id=34221” - “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=34221.

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)**, based on the decrypted query received at the server, device related commands are sent to the Local hub i.e., “to turn LAMP OFF”. The server processes the received result set 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
{
  "id" : "34221",
  "state" : "OFF",
  "level" : 80.0,
  "user_id" : "7IT6aT02cThiDA0pHshvX9e8z9t1",
  "hub_address" : "xxx.xxx.xxx",
  "pin" : 27,
  "type" : "LAMP"
}

```

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).

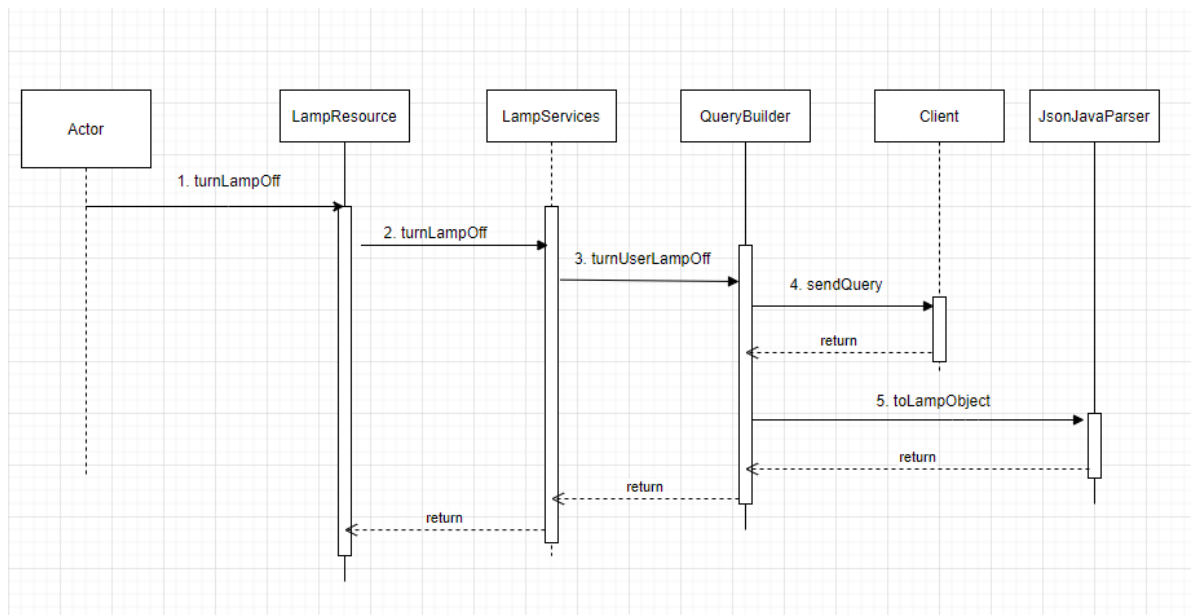


Figure 6 - Sequence diagram to turn off user's lamp at API level

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.

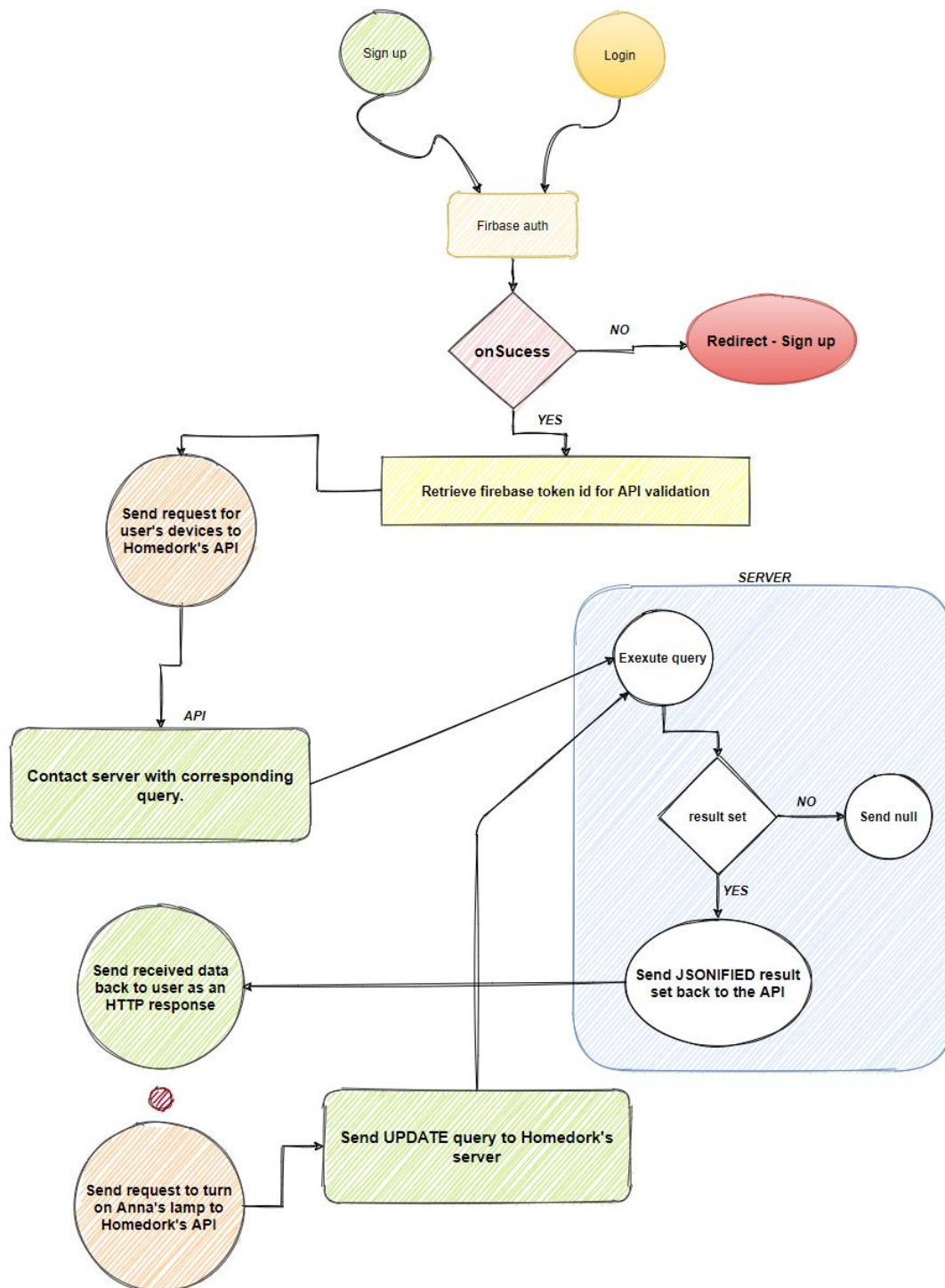


Figure 7 - Communication flowchart

D4 Sequence Diagrams

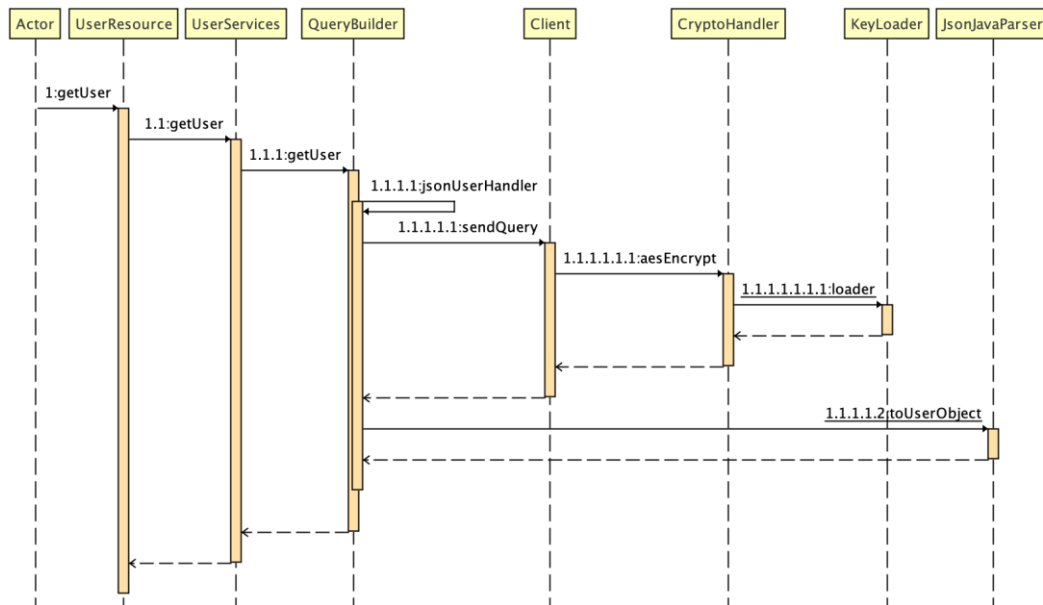


Figure 8 - Get user

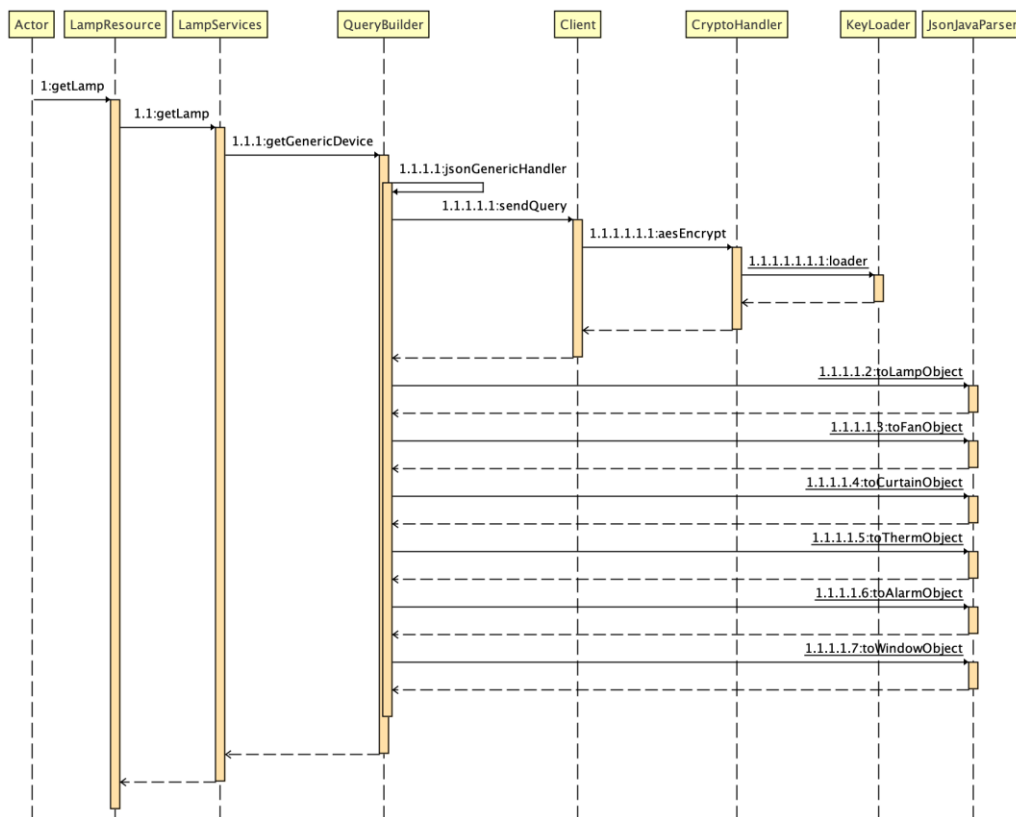


Figure 9 - Get device (get Lamp)

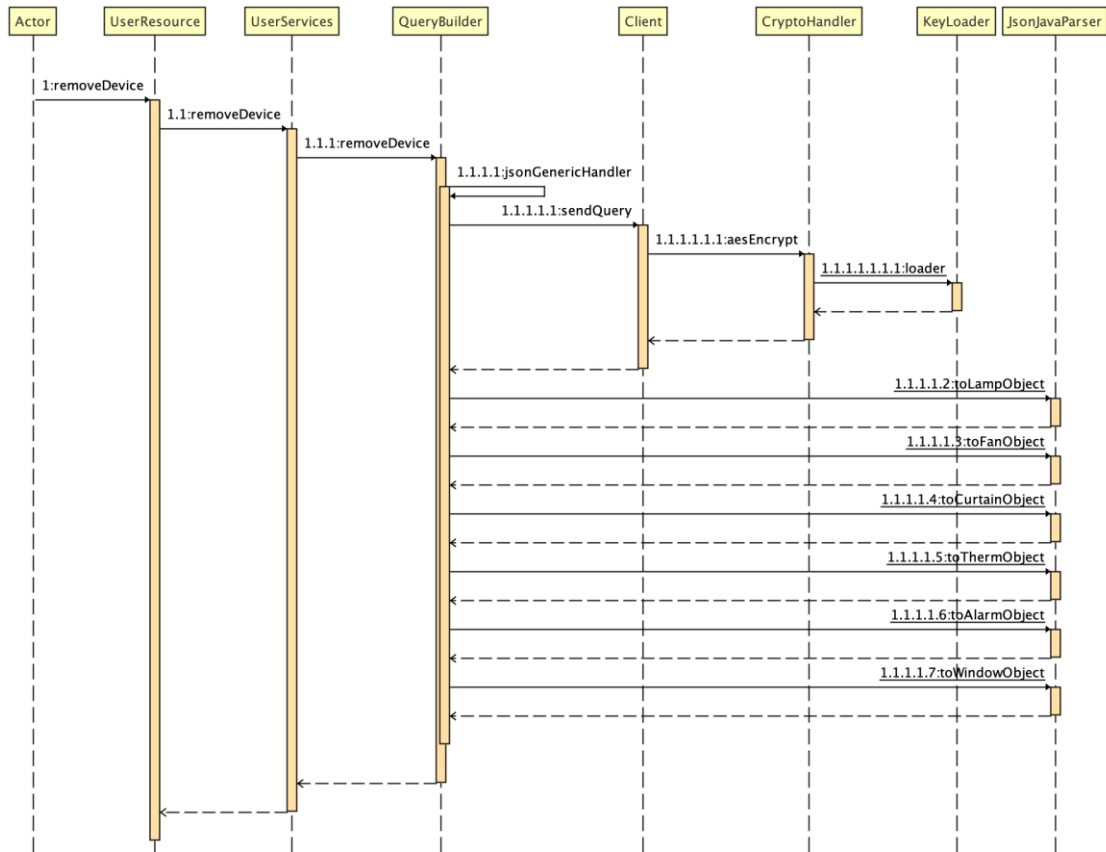


Figure 10 - Remove Device

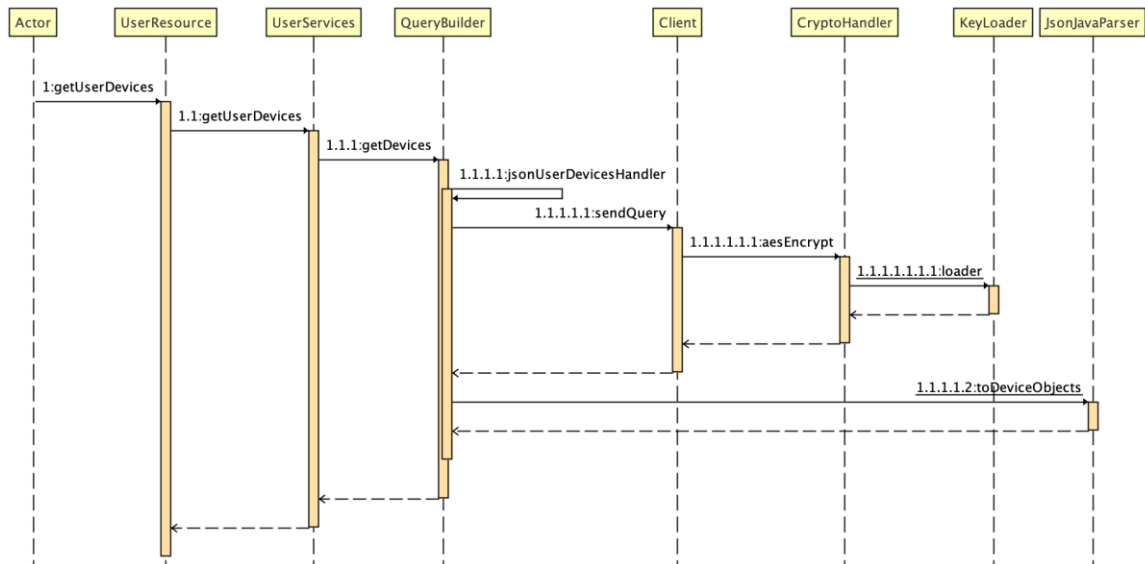


Figure 11 - Get user devices

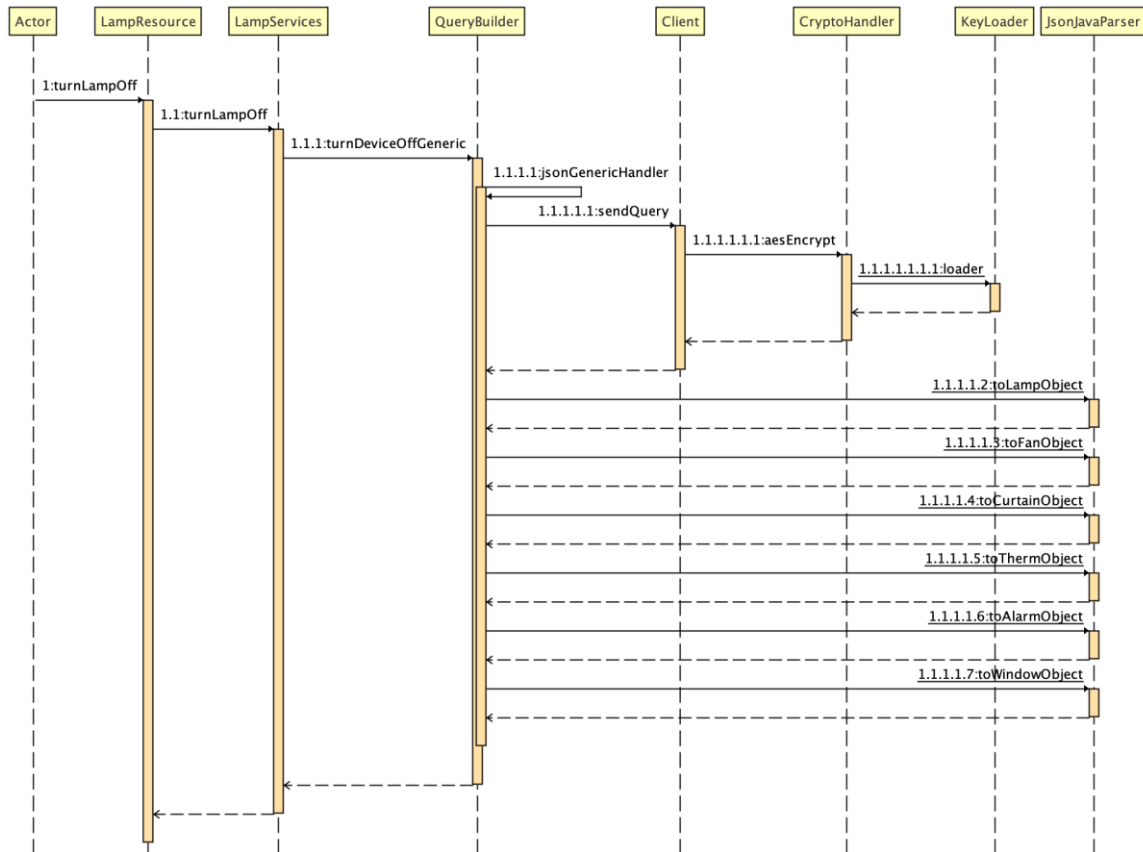


Figure 12 - Turn device off (turn lamp off)

D5 Class Diagrams

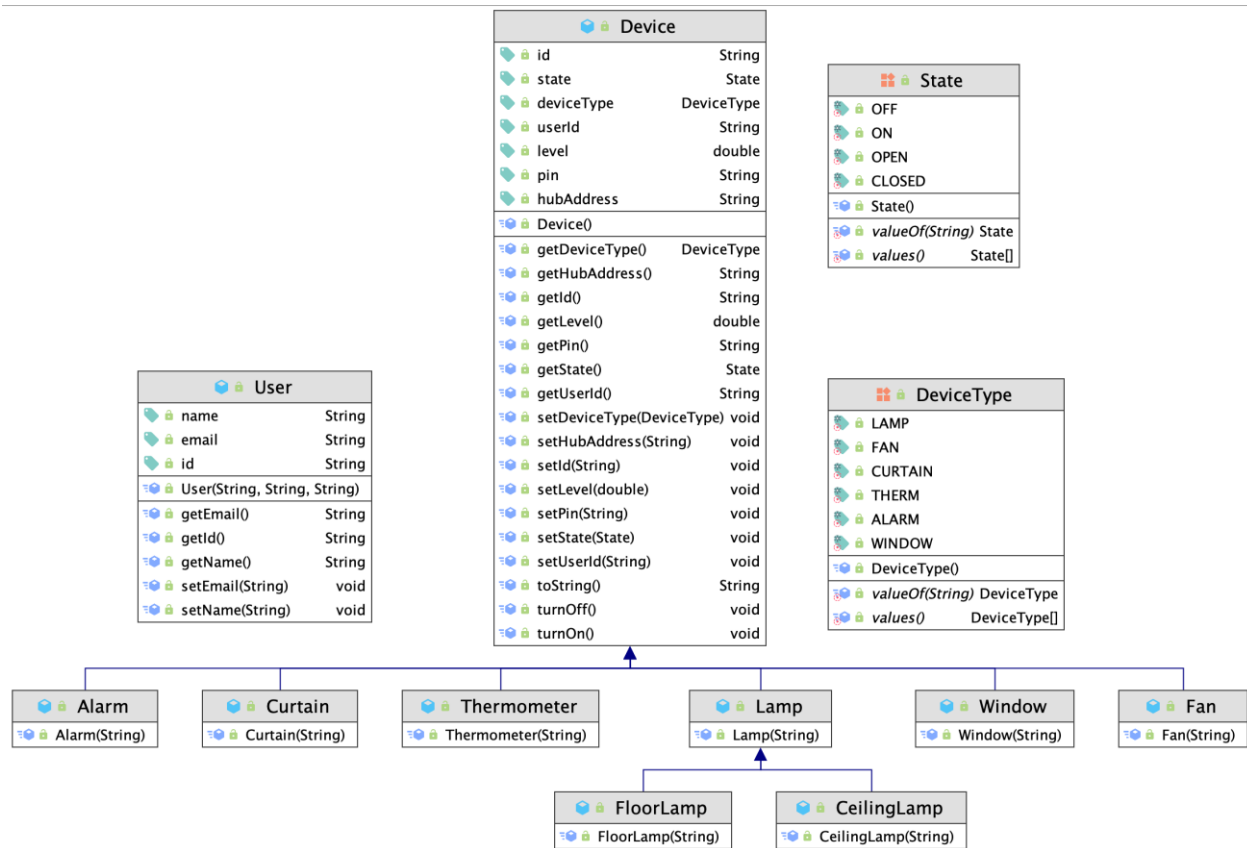


Figure 13: Model class Diagrams

Figure 13 shows 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 can 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 types 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.

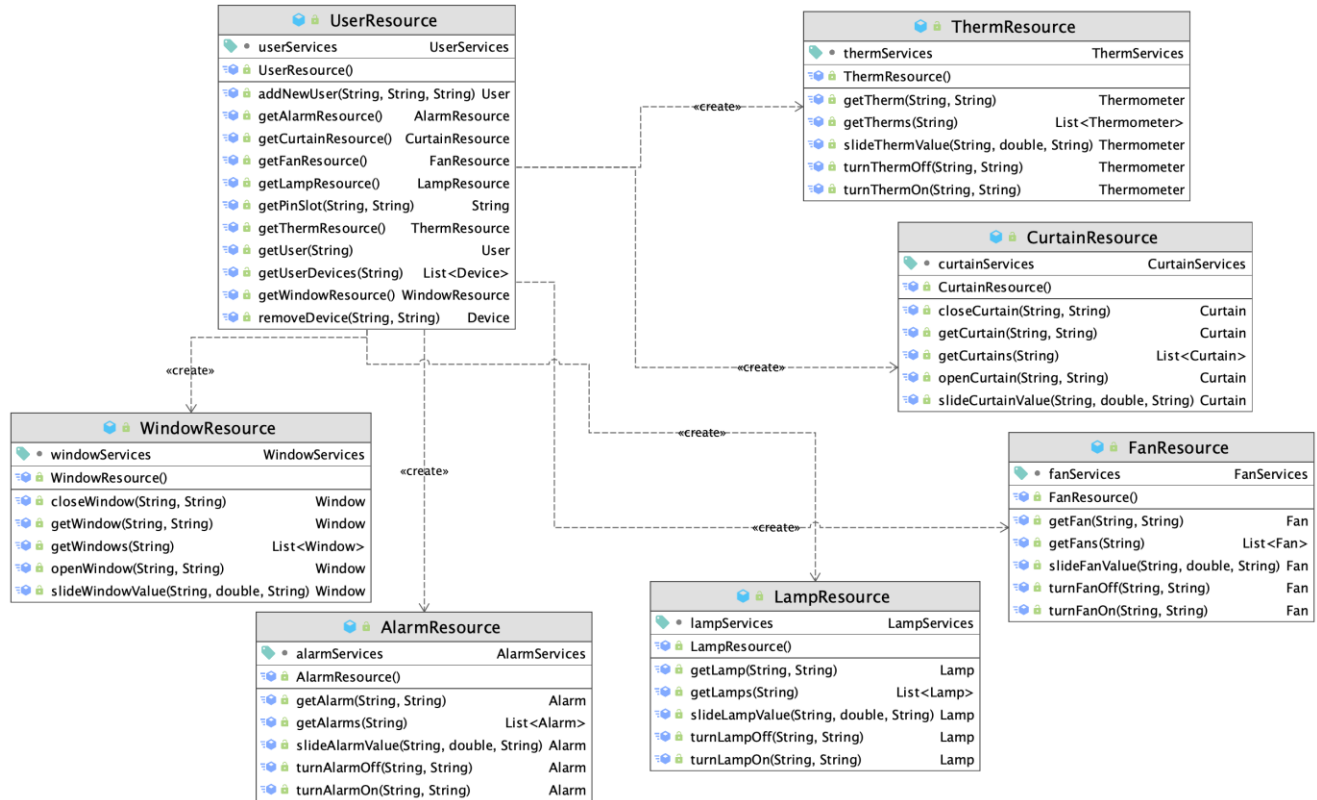


Figure 14 - API resource package and classes

Homedork has the “**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**).

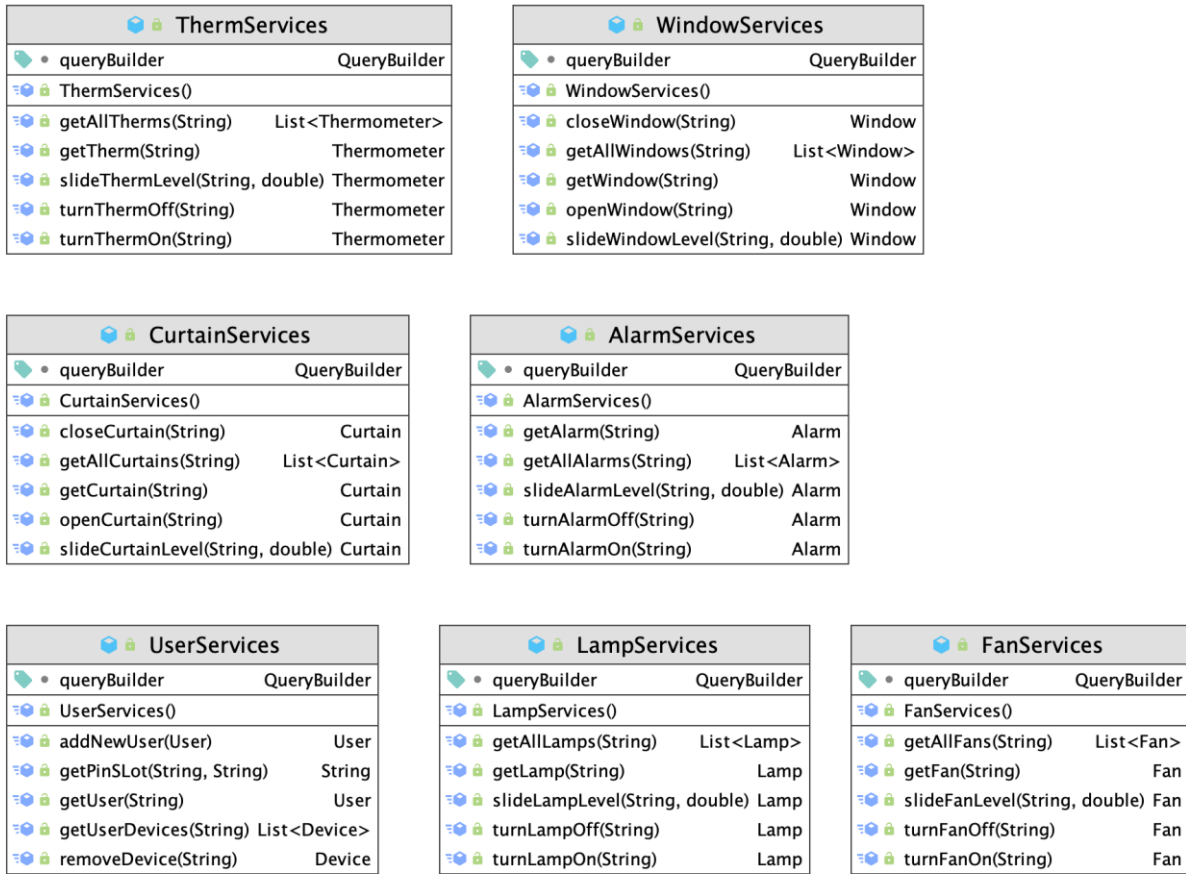


Figure 15 - Service classes diagram for all corresponding resource classes

Figure 15 shows all the implemented service classes and methods we have for all models. Each service class is used in its corresponding resource class, it basically handles calling specific query building methods used to handle user received HTTP requests.

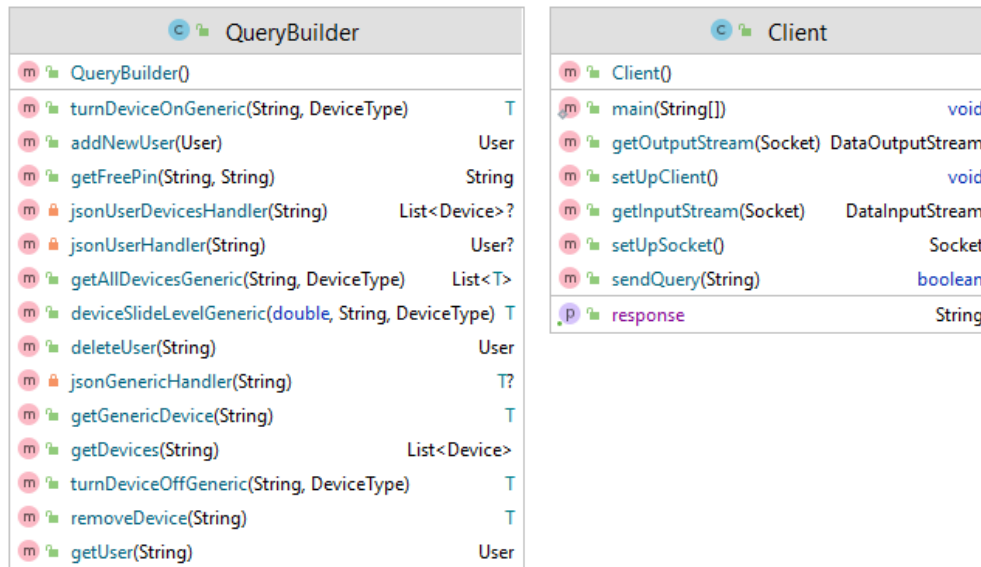


Figure 16 - Client and Query Builder class diagrams

Figure 16 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 string object** method. It is then parsed by a custom “JSONJavaParser” class using the GSON lib to its appropriate java object.

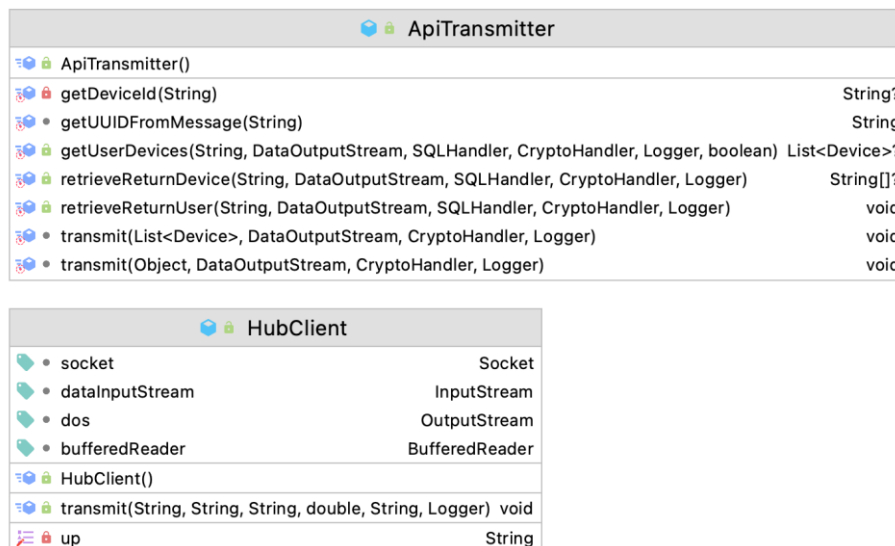


Figure 17 - Hub client and API transmitter class server

Figure 17 shows both classes responsible for communicating with the local hub and API using sockets. The API transmitter class is used to fetch responses to received requests, returning requests devices object, user object in the case of a login/sign up, returns a device object after an update process.

The Hub client class, connected with the local hub of the updated device using socket, getting the hub address from the result set of the update device, thus informing the local hub on whatever CRUD operation took place.

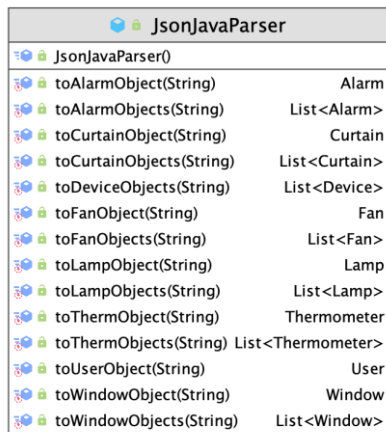


Figure 18 - JSONJavaParser class diagram

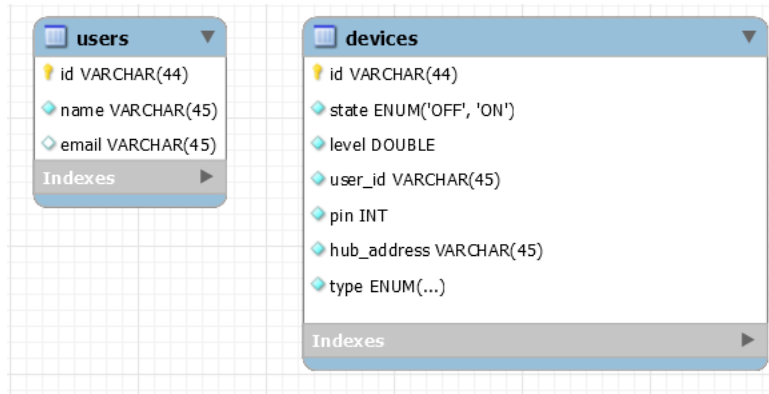


Figure 19 - Database tables

Figure 18 is the **JsonJavaParser** class that parses json object responses received from the database server into java objects.

Figure 19 shows the tables we have in our already made MySQL database. Each device saves the owner's user's id, the hub address and pin the device is connected to and device TYPE. This method provides a scalable solution in case of future works which would require more than 1 local hub connected to the server.