



DISTRIBUTED COMPUTING

Project 2: Web service

Marc Cervera Rosell - 47980320C

December 2022

Bachelor's degree in computer engineering

Contents

UML	3
Endpoints, operations, resulting codes and representations	3
Endpoint /home	3
POST operation	3
PUT operation	1
DELETE operation	1
GET operation	1
Endpoint /home/prts/ <criteria >/ <value >	2
Endpoint /sensor	2
POST operation	2
DELETE operation	3
Endpoint /home/ <homeid >/sensor	3
Endpoint /homeowner	3
GET operation	3
POST operation	4
Endpoint /homeowner/check/ <home_ownerid >	4
Endpoint /homeowner/ <home_ownerID >/homes	4
Endpoint /alldata	5
GET operation	5
POST operation	5
Endpoint /sensor/ <sensorID >	5
Representations	6
Some project justifications	8
Why Flask?	8
Why MySQL?	8
Why Swagger?	9
Execution cases	9
WS demo's first step	9
WS demo's second step	11
WS demo's third step	12
WS demo's fourth step	12
WS demo's fifth step	13
WS demo's sixth step	14
WS demo's seventh step	15
WS demo's eighth step	15

WS demo's ninth step	16
WS demo's tenth step	17
WS demo's eleventh step	18
WS demo's twelfth step	18
WS demo's thirteenth step	19
WS demo's fourteenth step	19
WS demo's fifteenth step	20
Source code	21
Hours dedicated to the project	21

List of Figures

1	UML digram of the database	3
2	Swagger's default view	6
3	Swagger's default view with a method specification displayed	7
4	Swagger's schemas	7
5	Creation of the first house	10
6	Creation of the second house	10
7	Creation of the third house	11
8	Update of the third house	11
9	Full search	12
10	House 2 deletion	12
11	Stoplight studio error while performing the DELETE operation	13
12	All houses in the system	13
13	Sensor 1 creation	14
14	Sensor 1 creation	14
15	Sensors in the first house	15
16	User 1 creation	15
17	User 2 creation	16
18	Esteban's house creation	16
19	Abel's house creation	17
20	Abel's house deletion	17
21	Partial search	18
22	Esteban's information	19
23	Result of checking an ID of an existing user	20
24	Result of checking a non-existing ID	20

UML

The UML diagram that corresponds to the used data model to solve this practical case, looks as It can be seen in the following image:

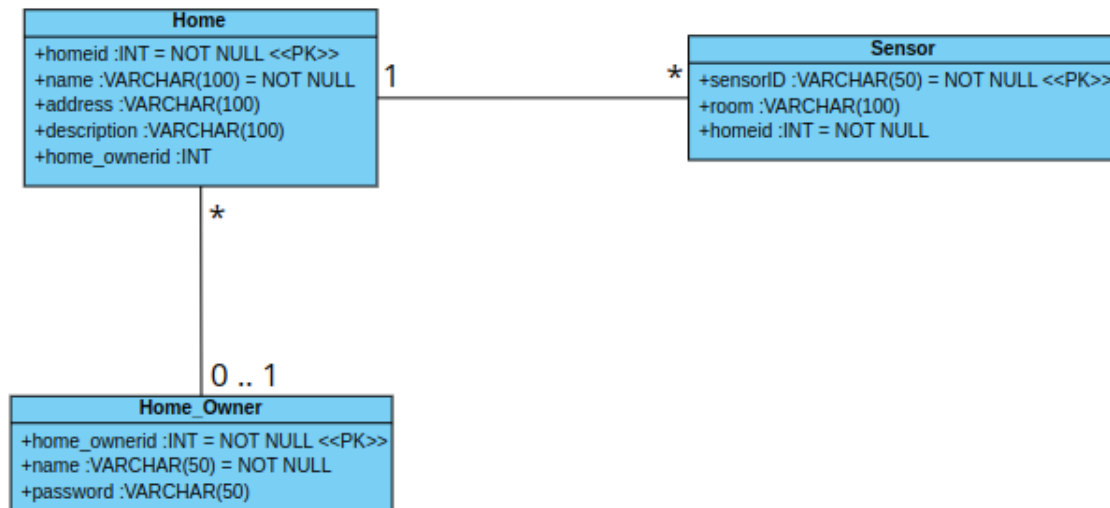


Figure 1: UML diagram of the database

Endpoints, operations, resulting codes and representations

Endpoint /home

This endpoint, has a total of four operations: POST, PUT, DELETE and GET.

POST operation

This operation will, firstly, get the data from the body petition and will validate it according to the *home_schema* schema. If the validation goes wrong, will raise a 400 error and an information message.

In case the data validation goes well, the second step of this operation will be, to check if the body petition contains a *home_ownerid*. In case, the body does not have a value for this parameter, the program will assign a default value of 1. Otherwise, will get it from the message.

Finally, the program will insert in the database the data. If the insertion is successful, a 201 code will be raised jointly with a message. In case of unsuccessful insertion, a 500 code will be raised jointly with an error message.

The insertion SQL query looks like:

```
INSERT INTO Home (name, address, description, home_ownerid) VALUES
('name', 'address', 'description', 'home_ownerid')
```

PUT operation

As in the above operation, the first step is to get the information from the petition body and validate it. If the validation is unsuccessful, a 400 code will be raised jointly with an error message.

In case of a successful validation, the next step will be to check the parameters of the petition body. First of all, the program will check if the body petition contains an update for the *name* field. If it contains it, will update the SQL query and if there's anything more to update, a comma will be added to the SQL query. This procedure, will be repeated for all the schema parameters.

The last step, will be to check if there's actually something to update in the database. If there's actually parameters to update, the program will perform the *UPDATE* SQL operation. If the update is successful, a 200 code will be raised jointly with a message. Otherwise, a 500 code will be raised jointly with an error message. Finally, if after all the checks, there's nothing to update, a 400 code will be raised jointly with a message.

DELETE operation

This is the penultimate operation of this endpoint.

As is usual, the first step is to get the information to delete. Once this data is obtained, the SQL *DELETE FROM* operation will be performed with the obtained data. If the SQL operation is successful, a 200 code will be raised jointly with a message. Otherwise, a 500 code will be raised.

The SQL query looks like:

```
DELETE FROM Home WHERE homeid = 'specified_homeid' and home_ownerid =
specified_home_ownerid
```

GET operation

This is the last operation of this endpoint.

To perform this operation, there are two possibilities. If there's nothing specified, all the houses will be listed. Otherwise, that is, a criteria search and a criteria value are specified, the SQL query will be conformed with this data. And the house that meets the parameters will be listed.

In case there isn't any house that meets the parameters, or there's not any house in the

database, a 500 error will be raised jointly with an error message. Otherwise, a 200 code will be raised jointly with the result of the query. The SQL query to list all the houses looks like:

```
SELECT * FROM Home
```

And the SQL query to list a specified home looks like:

```
SELECT * FROM Home WHERE criteria_search = criteria_value
```

Endpoint `/home/prts/ <criteria >/ <value >`

This endpoint, only has one GET operation. This operation performs the partial search. The operation will list all the data of a house that meets a criteria search and a pattern of the value that is associated with the criteria search.

```
SELECT * FROM Home WHERE criteria_search LIKE %pattern%
```

If, for any reason, the query is unsuccessful, a 500 code will be raised jointly with an error code. Otherwise, a 200 code will be raised jointly with the result of the query.

Endpoint `/sensor`

This endpoint has a total of two operations. These operations are: POST, DELETE.

POST operation

This operation's mission is to introduce new values in the database.

The first step is to get the data from the body and validate it according to the *sensor_schema* schema. If the validation is unsuccessful, a 400 code will be raised jointly with an error message.

If the validation becomes successful, the next step is to generate and execute the SQL query with the body data.

If the sensor creation query finishes successfully, a 201 code will be raised jointly with a message. Otherwise, a 500 code will be raised jointly with an error message.

```
INSERT INTO Sensor (sensorID, room, homeid) VALUES ('sensorID', 'room',  
            'homeid')
```

DELETE operation

This operation, works exactly in the same way as the DELETE operation of the */home* endpoint. The only difference is located in the SQL query. The */home* SQL DELETE query looks like:

```
DELETE FROM Home WHERE homeid = specified_homeid and home_ownerid =
specified_home_ownerid
```

And the DELETE operation of this endpoint looks like:

```
DELETE FROM Sensor s INNER JOIN Home h ON s.homeid = h.homeid WHERE
s.criteria_search = specified_criteria_search and h.home_ownerid =
specified_home_ownerid
```

The fact of performing the INNER JOIN operation between the table Home and the table Sensor is to make sure that a user can only delete his own sensors.

If the query execution finishes successfully, a 200 code will be raised jointly with a message.

Otherwise, a 500 code will be raised jointly with an error message.

The justification of using INNER JOIN instead of another JOIN type, is because if in one table does not exist the *homeid*, the JOIN operation will not work.

Endpoint */home/ <homeid >/sensor*

This endpoint, only has a GET method that allows us to list the sensor of a home.

This endpoint lists all the sensors of a concrete home. So, the *homeid* must be provided.

If the query finishes successfully, a 200 code will be raised jointly with the information.

Otherwise, a 500 code will be raised jointly with an error message. The SQL query looks like:

```
SELECT * FROM Sensor WHERE homeid = specified_homeid
```

Endpoint */homeowner*

This endpoint has a total of two operations. These operations are: GET and POST.

GET operation

The first step is to get the information from the body petition and validate it according to *homeowner_schema* schema. IF the validation is unsuccessful, a 400 code will be raised jointly with an error message.

Finally, this method will introduce in a SQL query the name of a homeowner and will

return all the data related with him/her. If the query finishes successfully, a 200 code will be raised jointly with the result of the query.

The SQL query looks like:

```
SELECT * FROM Home_Owner WHERE name = 'specified_name'
```

POST operation

This second, and last, operation of this endpoint, will firstly get the data from the body petition and will validate it according to the *homeowner_schema* schema. IF the validation is unsuccessful, a 400 code will be raised jointly with an error message.

So, in this case the SQL query will look like:

```
INSERT INTO Home_Owner (name, password) VALUES (name_value, password_value)
```

If the insertion is successful, a 201 code will be raised jointly with a message. Otherwise, a 500 code will be raised jointly with an error message.

Endpoint /homeowner/check/ <home_ownerid >

This endpoint, only has a GET operation. This operation will list all the information about a homeowner that has the specified ID.

Hence, the SQL query will look like:

```
SELECT * FROM Home_Owner WHERE home_ownerid = specified_home_ownerid
```

If the ID of the specified homeowner cannot be found in the database, a 404 code will be raised jointly with an error message. Otherwise, a 200 code will be raised jointly with a message.

If, for any reason, the query is “unable” to get the data (database connection), a 500 code will be raised jointly with an error message.

Endpoint /homeowner/ <home_ownerID >/homes

This endpoint only has a GET operation that will list all the homes from a homeowner. So, the SQL query will look like:

```
SELECT * FROM Home WHERE home_ownerid = specified_home_ownerid
```

As is usual, if the query finishes successfully, a 200 code will be raised, jointly with the result of the query and if, for any reason, the query fails, a 500 code will be raised jointly with an error message.

Endpoint /alldata

This endpoint, has a GET operation and a POST operation.

GET operation

This operation, lists all the homes from a homeowner.

In order to do it, the program will try to execute the following SQL query:

```
SELECT * FROM Home_Owner ho left join Home h on ho.home_ownerid =  
      h.home_ownerid left join Sensor s on h.homeid = s.homeid
```

If the query is successful, a 200 code is raised jointly with the result of the query. Otherwise, a 500 code is raised jointly with an error code. The justification of using the LEFT JOIN operation it's because in case a house does not have any sensors or a user does not have any houses, at least we can still get its user information.

POST operation

The first step, is to get the information from the body petition.

In this part, each insertion will have its own query. It's to say, there's going to be a query for the *Home_Owner* table insertion, another for the *Home* table insertion and another one for the *Sensor* table insertion.

In case there's nothing to insert into one of these tables, the SQL query will not be executed, due to each SQL query is inside an *if* statement that checks into which table we want to insert.

If all the queries, that have to be executed, finish successfully, a 200 code is raised jointly with a message. Otherwise, a 400 code is raised jointly with an error message.

Endpoint /sensor/ <sensorID >

This endpoint, only has a GET method.

This endpoint will connect to the MongoDB of the first project and will get the temperatures of the specified sensor.

Unfortunately, this method does not work, because I had to make changes in the first project before deliberating it and I haven't had enough time to implement this functionality.

Representations

To create the representations as in the movie database, the chosen tool has been Swagger. In order to be able to check the Swagger, you must:

1. Build and run the project.
2. In the browser, enter the following URL:

(a) <http://172.17.0.1:5000/api/docs/>

Even though the Swagger URL and the steps to watch the Swagger are provided. Let's take a look at some images of how It looks, the Swagger.

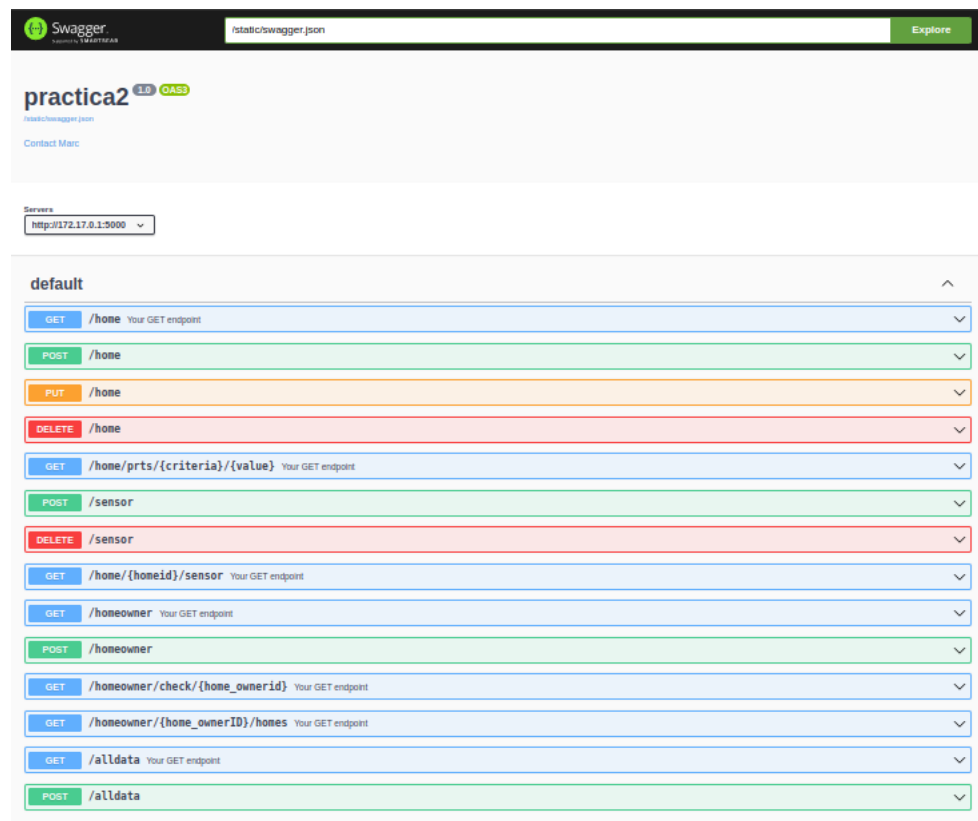


Figure 2: Swagger's default view

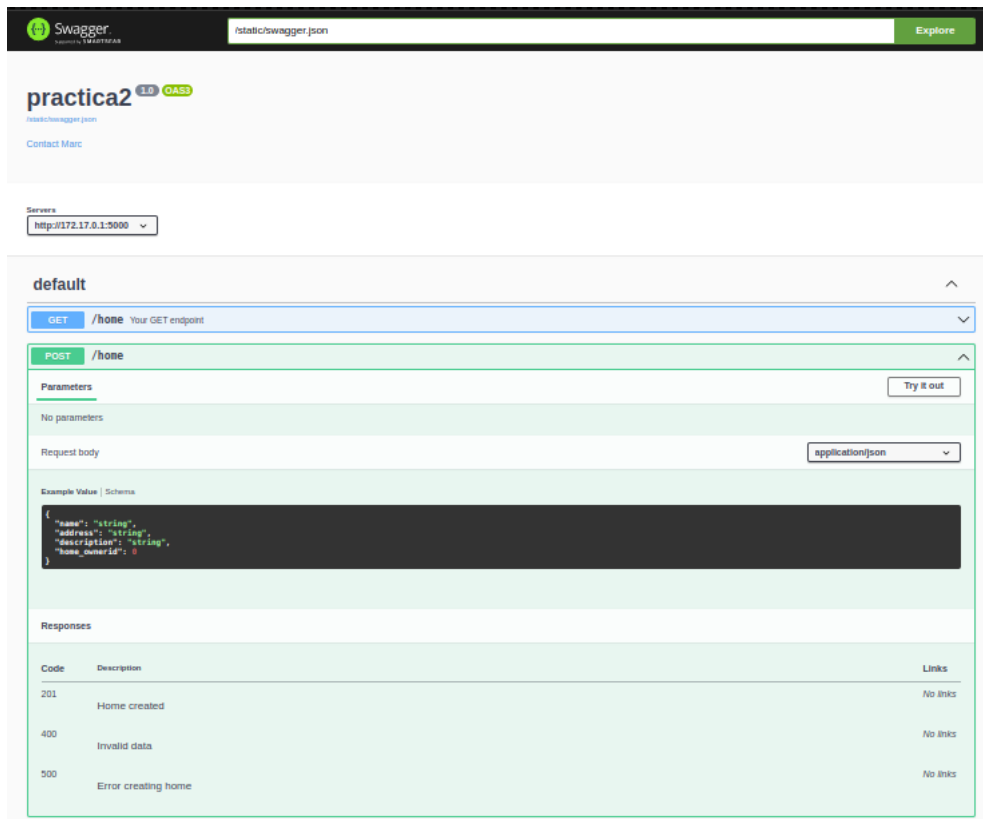


Figure 3: Swagger's default view with a method specification displayed



Figure 4: Swagger's schemas

If you want to see the Swagger in local, just enter in *apiREST/static/swagger.json*

Some project justifications

In this report's section are going to be justified the design decisions.

Why Flask?

Flask is a web framework, it's a Python module that lets programmers develop web applications easily. So, why is Flask a good choice? In addition, to all the Python advantages, Flask some advantages more.

The Flask's design is lightweight and modular. Therefore, it's easy to transform it into a web application or framework when one needs very few extensions without weighing much.

Flask is ORM-agnostic.

The basic foundation of API is very nicely shaped and made coherent.

The documentation of Flask is very comprehensive, filled with lots and well-structured examples. Users can even try out some sample applications to really get the real feel of Flask.

Flask can handle HTTP requests easily with the help of its functionalities.

It's highly flexible. Its configuration is even more flexible than that of Django, which gives its users plenty of solutions for every product they need.

Why MySQL?

MySQL is globally for being the most secure and reliable database management system used in popular web applications like WordPress, Drupal, Joomla, Facebook, and Twitter.

MySQL offers unmatched scalability to facilitate the management of deeply embedded apps using a smaller footprint, even in massive warehouses that stack terabytes of data. On-demand flexibility is the star feature of MySQL.

MySQL features a distinct storage-engine framework that facilitates system administrators to configure the MySQL database server for a flawless performance. MySQL is designed to meet even the most demanding applications while ensuring optimum speed, full-text indexes and unique memory caches for enhanced performance.

MySQL comes with the assurance of 24X7 uptime and offers a wide range of high availability solutions like specialized cluster servers and master/slave replication configurations. MySQL tops the list of robust transactional database engines available on the market. With features like complete atomic, consistent, isolated, durable transaction support, multi-version transaction support, and unrestricted row-level locking, it is the go-to solution for full data integrity. It guarantees instant deadlock identification through server-enforced referential integrity.

With the average download and installation time being less than 30 minutes, MySQL means usability from day one. Whether your platform is, MySQL is a comprehensive solution with self-management features that automate everything from space expansion and configuration to data design and database administration.

Why Swagger?

Swagger provides a unique and convenient platform to document, test, and write API structures.

Swagger provides a method to automate the documentation, which means Swagger picks up the methods with GET, PUT, POST, DELETE attributes and prepares the documentation by itself. Further, if any changes are implemented, then the Swagger documentation is automatically updated.

Swagger provides a UI integrated page where all the API methods are listed and enables the user to test any method that is required from the UI.

Swagger does the documentation in a conventional way (OpenAPI) which means it is in a machine-readable language. If a user starts the documentation first, Swagger will write the structure of the API automatically based on the written documentation. The API logic relies on the developer and business requirements, but the structure will be written by Swagger itself.

The user does not need separate applications to test APIs. Just configure Swagger once in the project and access it through a URL to test the APIs.

Swagger provides immense support for a wide range of platforms, languages, and domains.

Execution cases

In order to demonstrate the correct functioning of the API, I'm going to use the *Stoplight Studio* application, and I'm going to follow the steps of the *WS demo* section of the statement.

Before starting with the demos, it must be said that the IDs will not start in "normal" indexes because of previous tests.

WS demo's first step

This first step requires creating three different houses, with the two first ones with a similar description.

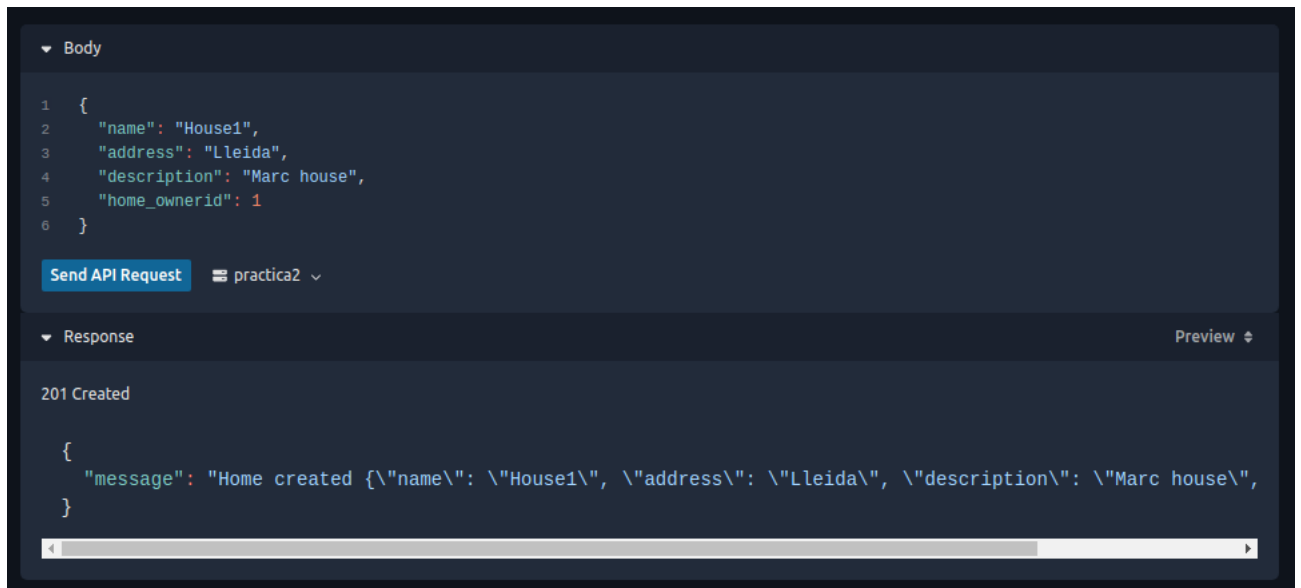


Figure 5: Creation of the first house

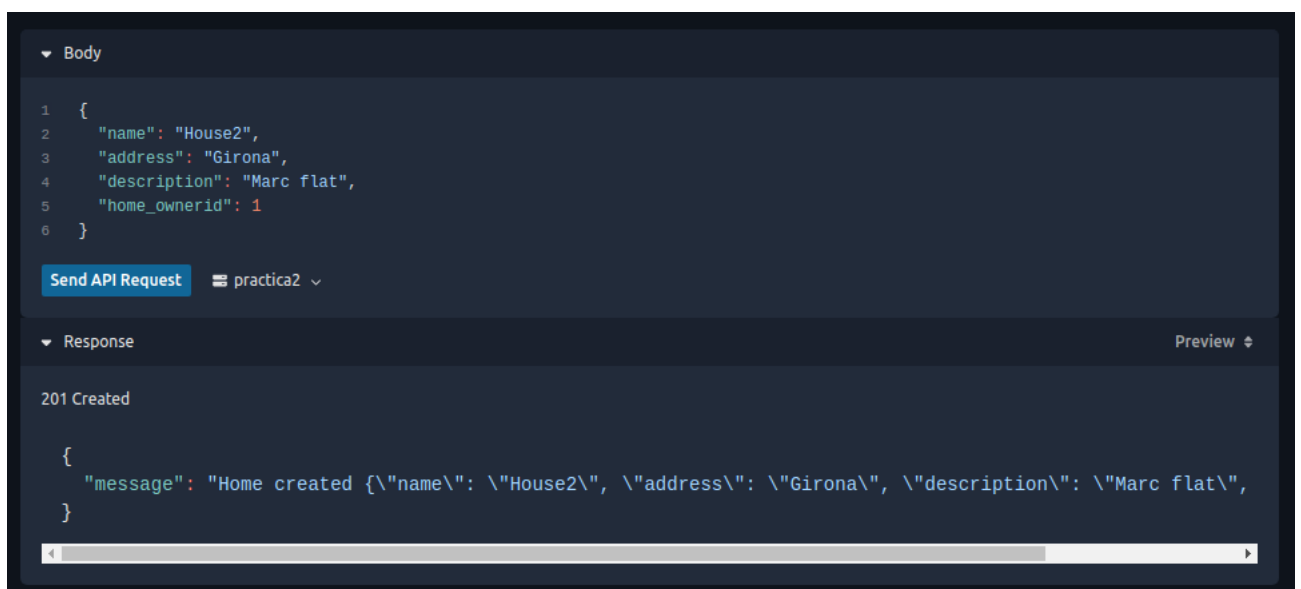


Figure 6: Creation of the second house

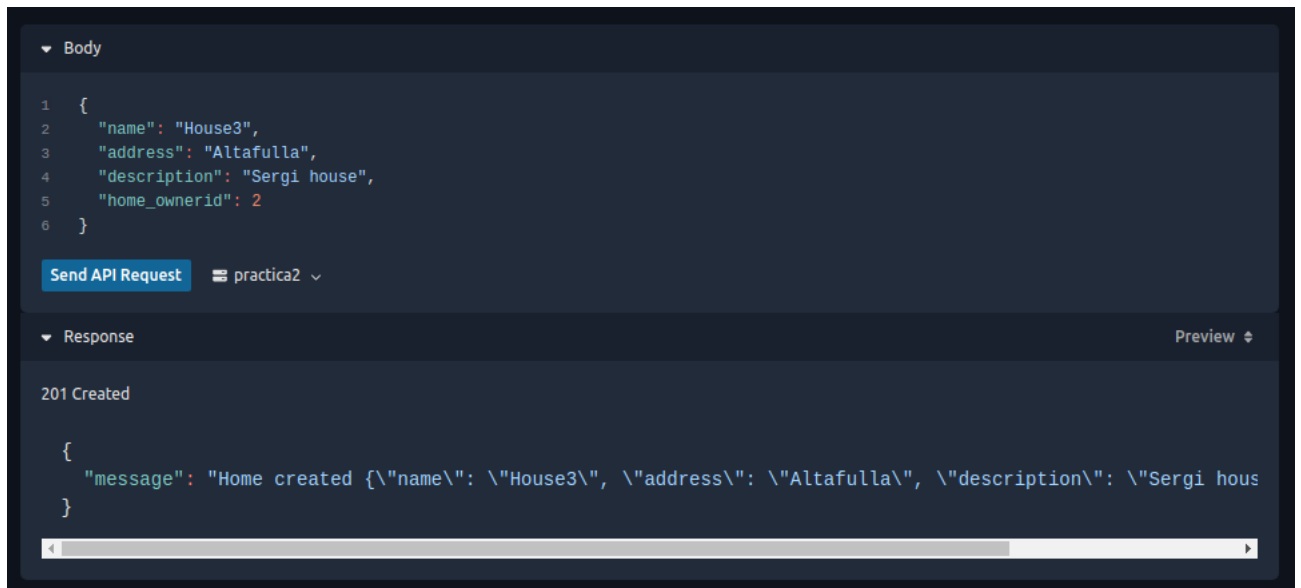


Figure 7: Creation of the third house

As it can be seen, every request has become successful due to a 201 has been raised jointly with the information that has been introduced.

WS demo's second step

This step, requires modifying the description of the third house.



Figure 8: Update of the third house

As it can be seen, the third's house description update, is successful because a 200 code is raised.

WS demo's third step

This step requires searching a home by its description.

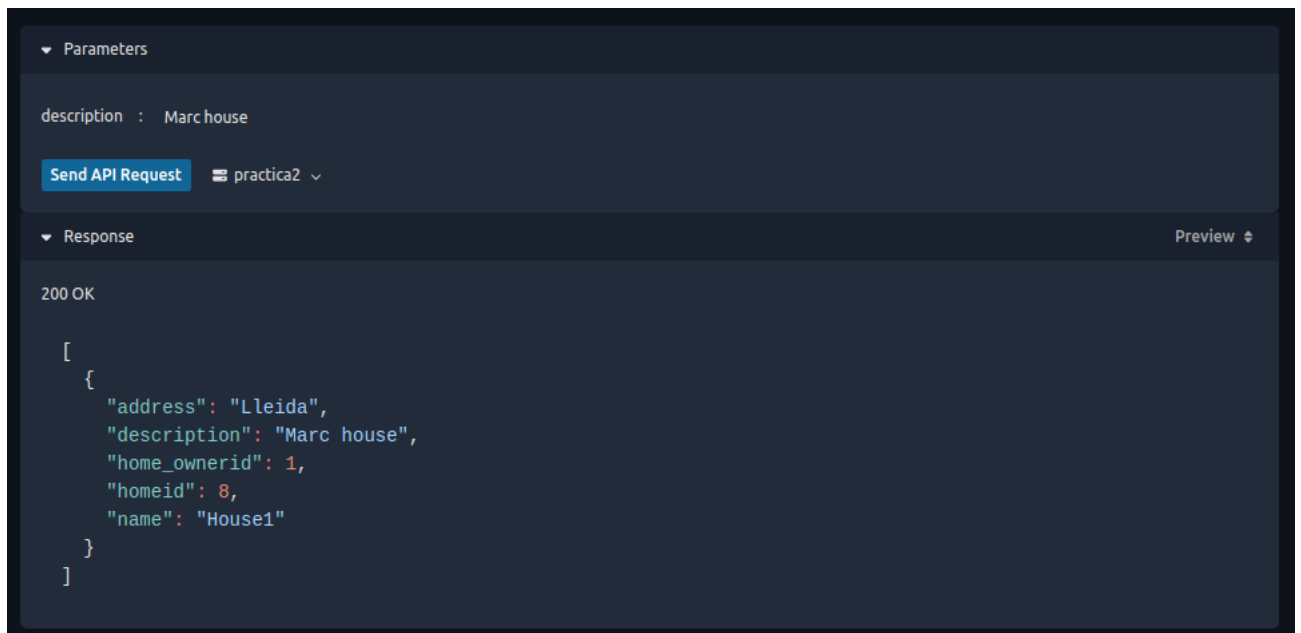


Figure 9: Full search

As it can be seen, the operation finishes successfully, due to a 200 code is raised.

WS demo's fourth step

This fourth step requires deleting the second home.

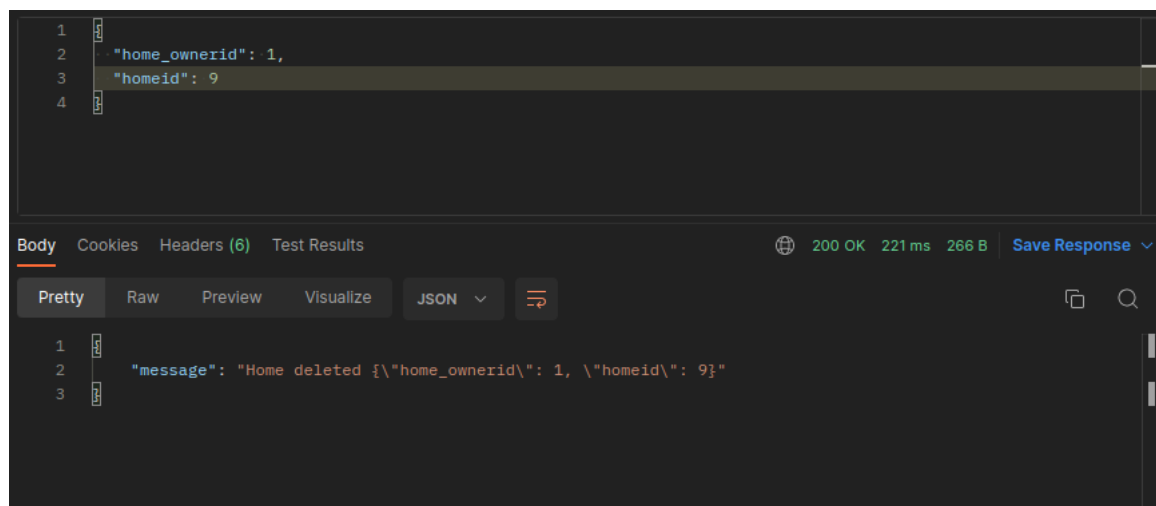


Figure 10: House 2 deletion

As it can be seen, the DELETE operation performs normally due to a 200 code is raised jointly with an informative message.

This operation has been performed with Postman because, in *Stoplight studio*, for some reason does not work.



Figure 11: Stoplight studio error while performing the DELETE operation

It seems that, for some reason, does not get the data.

WS demo's fifth step

This section of the demo demands to list all the houses

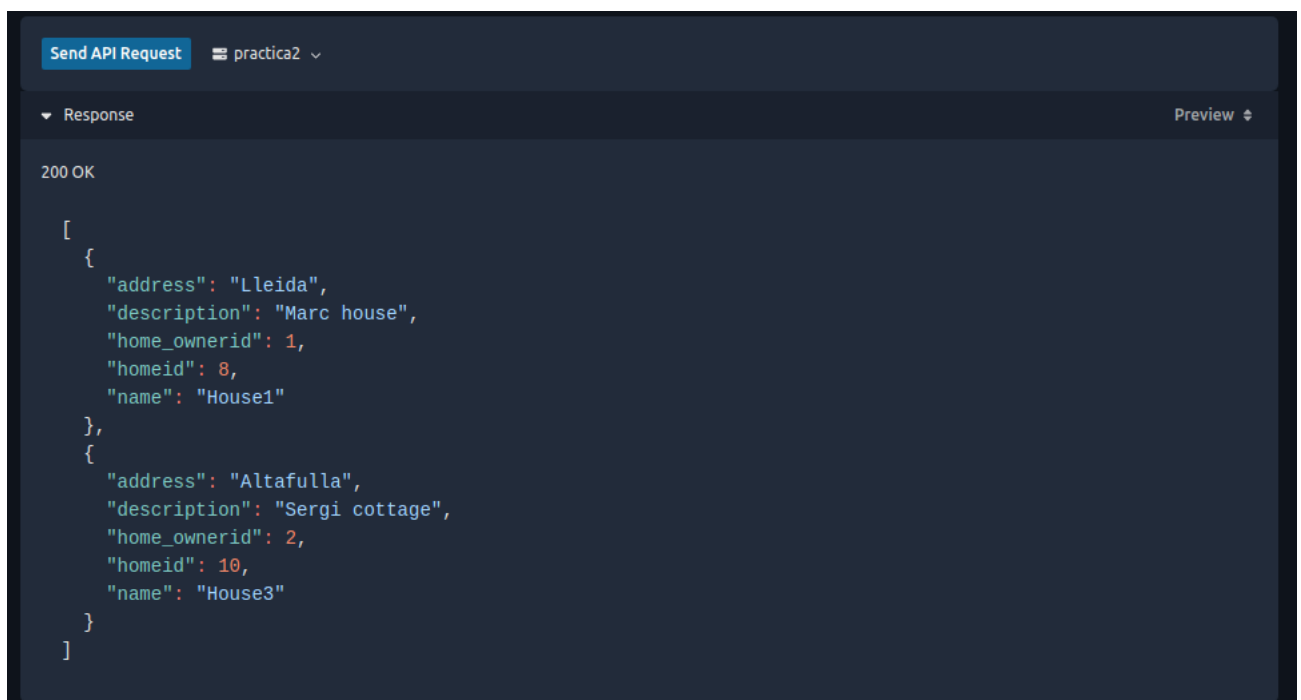


Figure 12: All houses in the system

As it can be seen, the operation finishes successfully, due to a 200 code is raised.

WS demo's sixth step

This part of the demo, requires creating two sensors for the first home.



Figure 13: Sensor 1 creation



Figure 14: Sensor 1 creation

As it can be seen, the creation of both sensors finish successfully due to a 201 code is raised.

WS demo's seventh step

In this part, the statement requires listing all the sensors in the first home. In this case, list all the sensors will coincide with the sensors that have just been created.

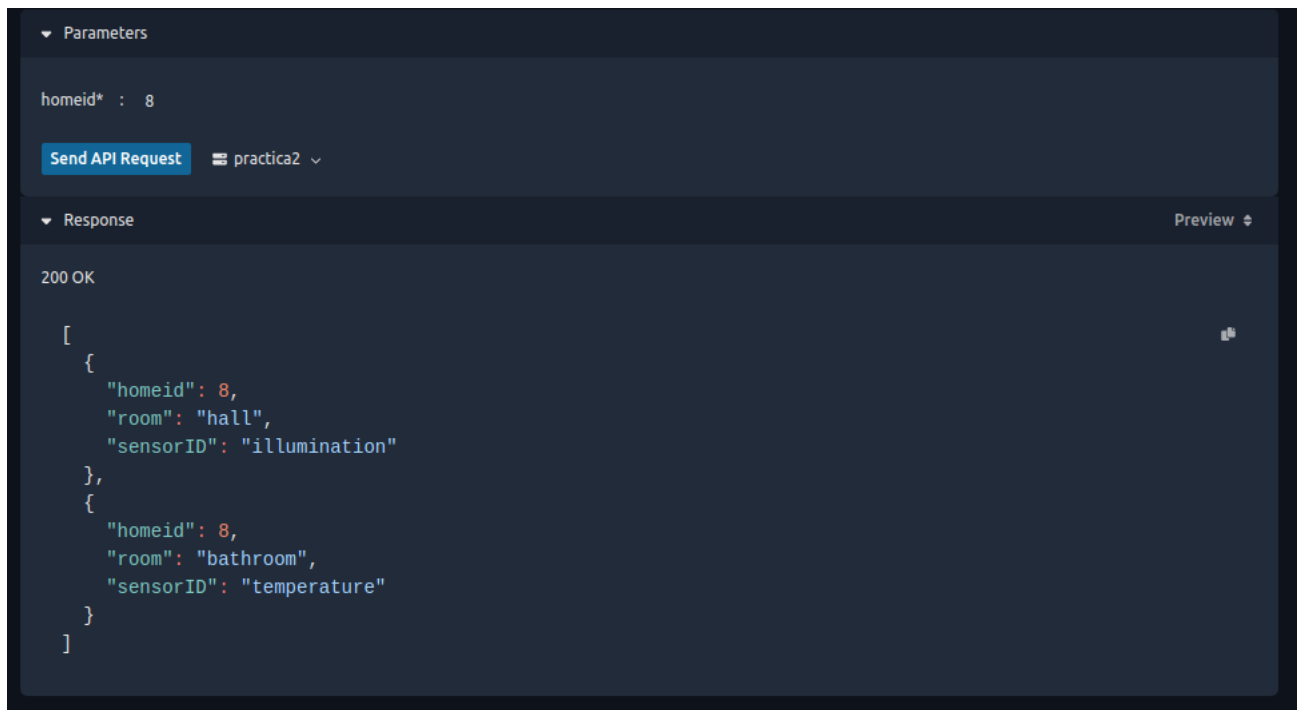


Figure 15: Sensors in the first house

As it can be seen, the GET operations work well due to a 200 code is raised.

WS demo's eighth step

This section demands creating two different users.



Figure 16: User 1 creation



Figure 17: User 2 creation

As it can be seen, the users are created successfully due to a 201 code is raised.

WS demo's ninth step

In this part, we've to assign a new house to each new user.

If we take a look in the *Home_Owner* table, we will see that the user *Esteban* has an ID of 4 and the user *Abel* has an ID of 5. So, when creating the houses, the field *home_ownerid* will have a value of 4 or 5.

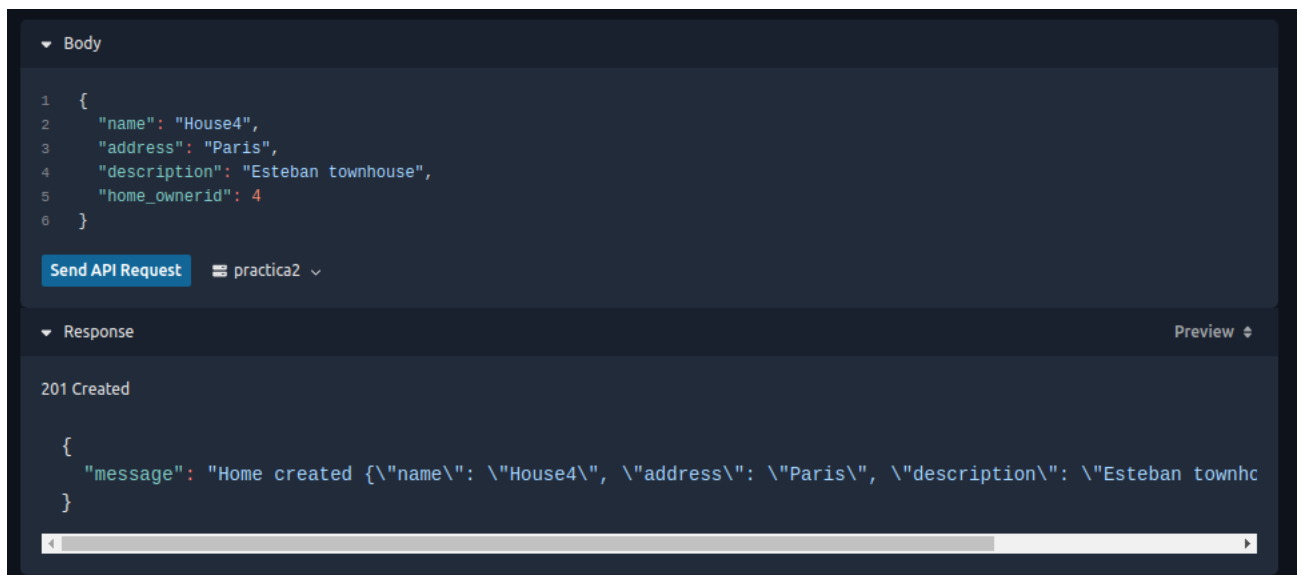


Figure 18: Esteban's house creation

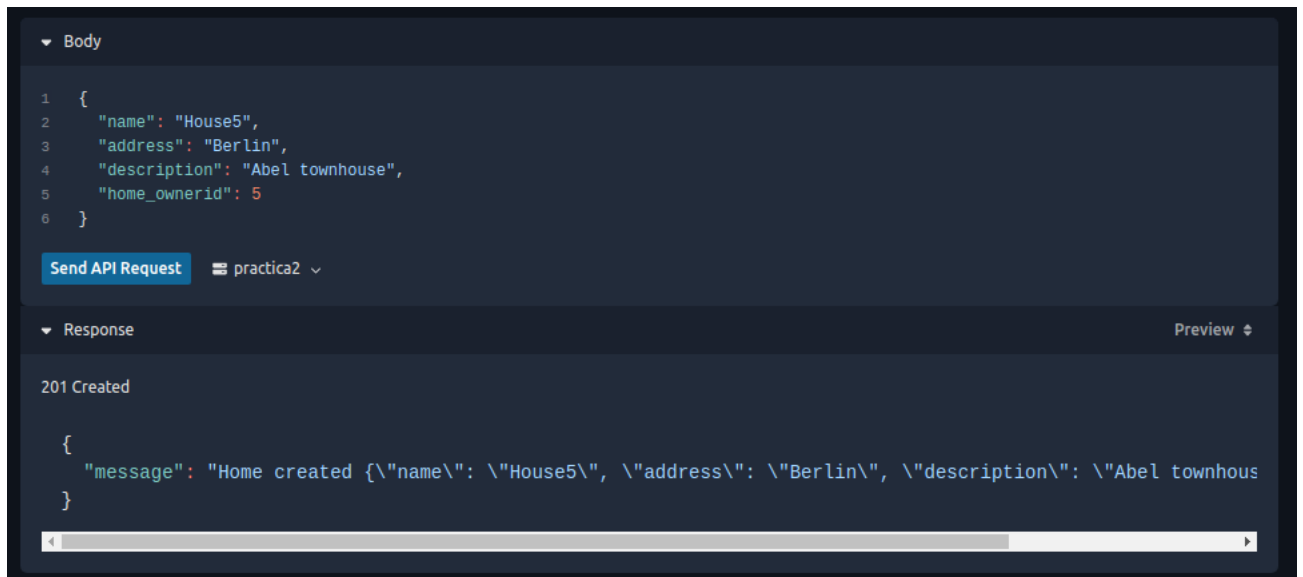


Figure 19: Abel's house creation

The operation finishes successfully due to a 201 code is raised.

WS demo's tenth step

Now, the house of the second created user (Abel) has to be deleted.

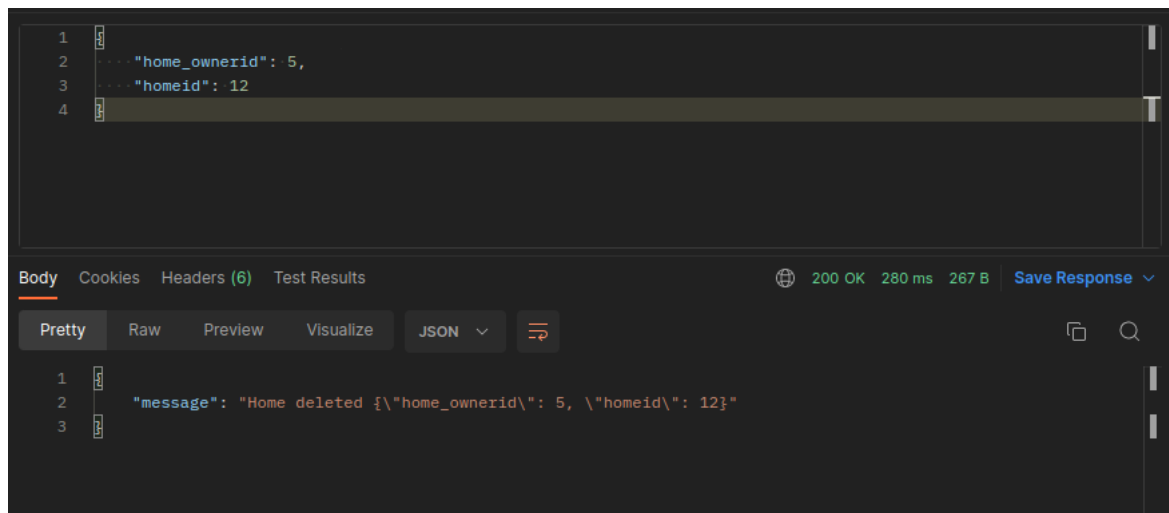
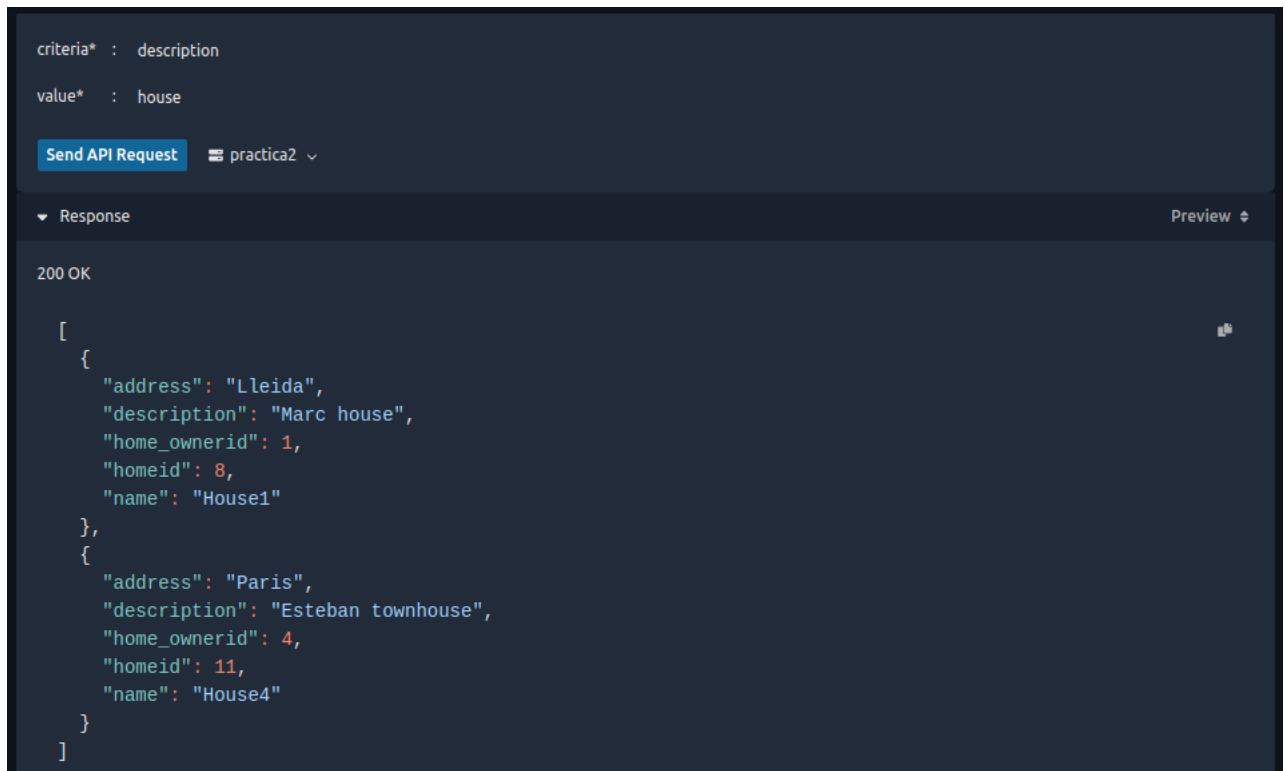


Figure 20: Abel's house deletion

As it can be seen, the operation is performed in Postman for the same reason as the last DELETE operation, and as it can be seen too, the deletion finishes successfully due to a 200 code is raised.

WS demo's eleventh step

Now, a house has to be searched (by its description) another time, but this time, the description is not going to be provided fully.



The screenshot shows an API client interface with a dark theme. At the top, the request configuration is displayed: `criteria* : description` and `value* : house`. Below this is a blue button labeled "Send API Request" and a dropdown menu showing "practica2". The response section is expanded, showing a "200 OK" status. The response body is a JSON array containing two house objects. The first object has "address": "Lleida", "description": "Marc house", "home_ownerid": 1, "homeid": 8, and "name": "House1". The second object has "address": "Paris", "description": "Esteban townhouse", "home_ownerid": 4, "homeid": 11, and "name": "House4". A "Preview" link is visible in the top right of the response area.

```
criteria* : description
value* : house

Send API Request practica2

Response Preview
200 OK

[
  {
    "address": "Lleida",
    "description": "Marc house",
    "home_ownerid": 1,
    "homeid": 8,
    "name": "House1"
  },
  {
    "address": "Paris",
    "description": "Esteban townhouse",
    "home_ownerid": 4,
    "homeid": 11,
    "name": "House4"
  }
]
```

Figure 21: Partial search

As it can be seen, a house can be searched by part of its description. This can be affirmed due to a 200 code is raised.

WS demo's twelfth step

Due to there're two users as the result of the partial search, only the user *Esteban* and its information will be shown.

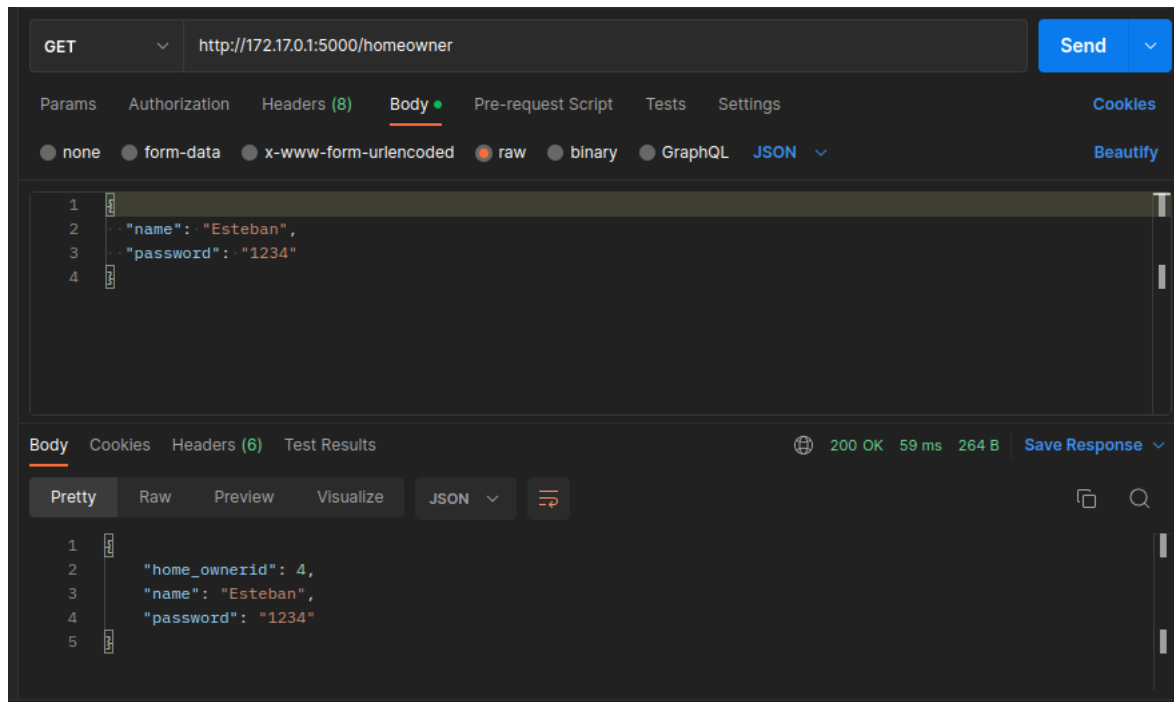


Figure 22: Esteban's information

As it can be seen, the operation finishes successfully due to a 200 code is raised.

WS demo's thirteenth step

Unfortunately, this method does not work, because I had to make changes in the first project before deliberating It and I haven't had enough time to implement this functionality.

WS demo's fourteenth step

In this penultimate part, an ID of an existing user must be checked.

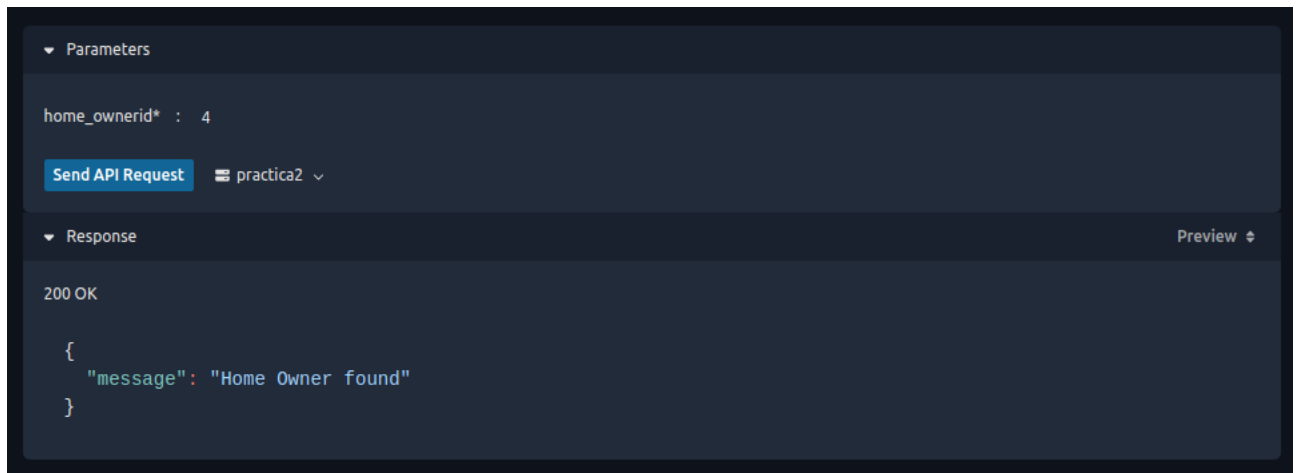


Figure 23: Result of checking an ID of an existing user

As it can be seen, a 200 code is raised jointly with an informative message that allows the users to know that the ID is present in the database.

WS demo's fifteenth step

Finally, a non-existing ID must be checked.

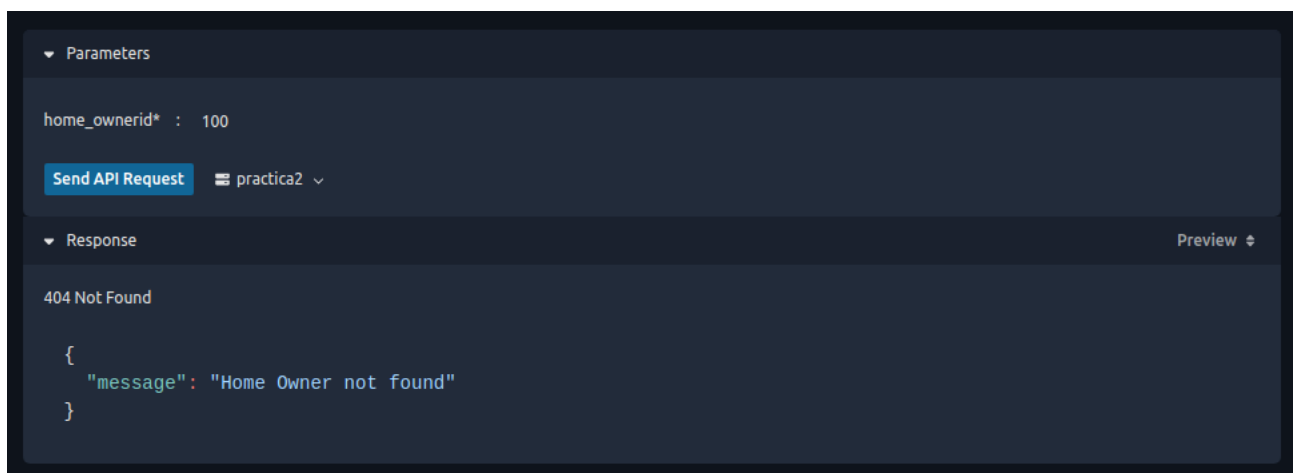


Figure 24: Result of checking a non-existing ID

As it can be seen, a 404 code is raised jointly with an error message informing that the user ID does not exist.

Source code

The code of the project can be found in a *.zip* file in the virtual campus' assignment, and can be found too in the following GitHub link: [GitHub repository](#). In order to see the whole project, you must be sure you're in the "main" branch and then, enter into the *Projects* folder and, finally, in the *Project 2* folder.

Hours dedicated to the project

The total hours dedicated to the project are:

- Programming part: More or less 50 hours.
- Testing part: More or less 20 hours.
- Documentation part: More or less 15 hours.