Instituto Superior Técnico

Mestrado em Engenharia Electrotécnica e de Computadores

# Aplicações Distribuídas Sobre a Internet

2021/2022 - 1ºSemestre, 1º Período

Relatório

**Grupo nº 21**

Nº 93047 - Diogo Madeira Martins

Nº 102247 - João Lucas Cohen Nascimento Rocheteau

Turno: ADIntL03

Data:14/11/2021

# 1 Architecture

- Displayed in the last page of the report

# 2 Endpoints

## 2.1 Service (Port: 8000)

### 2.1.1 - "/"

1. Method(s): GET.

2. Input data: Nothing.

3. Output data: redirect(authorization url) - Redirect the user/resource owner to the OAuth provider (IST Fenix in our case). We store the oauth state in a session variable.

### 2.1.2 - "/callback"

1. Method(s): GET.

2. Input data: The user has been redirected back from the provider to the registered callback URL. With this redirection comes an authorization code included in the redirect URL.

3. Output data: We will use the authorization code to get an access token that will be stored in a session. Redirects to /authsucess.

### 2.1.3 - "/authsucess"

1. Method(s): GET.

2. Input data: Token that was stored in a session.

3. Output data: In case a authenticated user is not on UserData database, we create one based on the OAuth provider information. Redirects to menu.

### 2.1.4 - "/menu"

1. Method(s): GET

2. Input data: Session's Token

3. Output data: If the session's token is equal to the User's database stored token , displays the menu webpage (menu.html). If this condition does not verify displays a error webpage instead (invalidtoken.html)

#### 2.1.5 - "/newgate"

1. Method(s): GET

2. Input data: Session's token and Session's istNumber (the user's ID)

3. Output data: If the session's token is equal to the User's database stored token, will present one of two webpages: The gate insertion webpage "newGate.html", in case the user is admin, or the menu webpage, in the case the user is not admin. If the token is wrong it renders a token error page instead.

#### 2.1.6 - "/gates/login"

1. Method(s): GET

2. Input data: Nothing.

3. Output data: Displays a simple form-type web page in order to be possible to start a chosen gate.

#### 2.1.7 - "/gates/newqrcode"

1. Method(s): GET

2. Input data: Session's token , Session's istNumber.

3. Output data: If the session's token is equal to the User's database stored token, it will present a new QR code so the user can authenticate at a gate. The code given is valid for 30s. If the token is wrong it renders a token error page instead.

#### 2.1.8 - "/gates/id/checkcode

1. Method(s): POST

2. Input data: Java Script's scanned QR Code in string format

3. Output data: Confirms if the code is in line with what's at the UserData. Returns -1 if it's not ; istNumber if it's valid.

#### 2.1.9 - "/gates/gateid/user/userid/confirmed"

1. Method(s): GET.

2. Input data: Gateid , istNumber

3. Output data: As the user istNumber could pass the the gateid gate - it updates both GateData's databases with one more access, creates a new instance of the stats of the gate and the stats of the user (calling the /gatestats and /user/id/stats endpoint). Displays a success web page (usercodegood.html) in which a green light gif is visable during the time the gate is opened, when it closes, it displays a red light.

#### 2.1.10 - "/gates/id/user/notconfirmed"

1. Method(s): GET.

2. Input data: Gateid

3. Output data: Creates a new instance of the stats of the gate (calling the /gatestats), showing that there was a failed attempt at opening the gateid gate. Displays an error page ('usercodebad.html').

#### 2.1.11 - "/gatestats"

1. Method(s): GET.

2. Input data: session's istNumber , session's token

3. Output data: Displays the stats of the gates in case the token is valid and the session's user is an admin. Output's a token error web page if the token is wrong. Redirects to the menu again if the user is not an admin.

1. Method(s): POST.

2. Input data: Status of the gate (0 - Not opened , 1 - Opened) , the date , gateid 3. Output data: Creates a new instance of gatestats with the information above. Retuns a JSON that had the input. data.

#### 2.1.12 - "/user/stats"

1. Method(s): GET

2. Input data: session's istNumber and session's token

3. Output data: If the token is valid, displays a webpage that shows the a determined istNumber user stats making use of /user/id/stats. If the token is invalid, displays a token error web page.

#### 2.1.13 - "/user/id/stats"

1. Method(s): POST

2. Input data: istNumber, date , gateid

3. Output data: Creates a new user stats instance with the information above. Retuns a JSON that had the input.

#### 2.1.14 - "/gates/login/confirm"

1. Method(s): POST

2. Input data: gate id ,gate location and gate secret from a form.

3. Output data: Checks if the information displayed in the form is correct, if it is, displays a camera in order to read the user's QRCode. If the gate information is incorrect, it displays a error web page (usercodebad.html).

#### 2.1.15 - "/gates"

1. Method(s): GET

2. Input data: Nothing.

3. Output data: A simple webpage featuring all the gates "listGates.html". This endpoint validates the session's token and is only acessed by admins.

1. Method(s): POST

2. Input data: Session's token and istNumber , gate's secret and gate's location

3. Output data: Checks if the information displayed in a form is correct, if it is and the current user is an admin, creates a new gate based in that information and displays a success page (gatecreated.html). If the gate information is incorrect, it displays a error web page (gatecodebad.html). This endpoint validates the session's token.

### 2.2 GateData and GateDataReplica (Port 7000 and Port 9000 respectively)

#### 2.2.1 - "/gates/code"

1. Method(s): GET

2. Input data: gate id

3. Output data: Returns a gate id's gate secret

4. Error codes: 400 - In case the request in not in the right format ; 404 if there's no gate that has the input gate id

#### 2.2.2 - "/gates/id/numb"

1. Method(s): PUT

2. Input data: gate id

3. Output data: updates the number of activations of the gateid's gate. Returns a not important string.

#### 2.2.3 - "/gatestats"

1. Method(s): POST

2. Input data: status, date and gate id

3. Output data: Creates a new gate stats instance with the information provided before. Retuns a not so important string.

4. Error codes: 400 if the request was not well formatted. Rollback's if the session didnt commit.

1. Method(s): GET

2. Input data: Nothing.

3. Output data: Returns a JSON with all the stats of the gates.

4. Error codes: 404 if the there is no gates with stats.

#### 2.2.4 - "/gates"

1. Method(s): POST

2. Input data: gate location , gate secret

3. Output data: Creates a gate based on the input information. Returns a JSON delivering the 'File created successfully' message.

4. Error codes: 400 if the request was not well formatted.

1. Method(s): GET

2. Input data: Nothing.
3. Output data: List all the gates information on a JSON.
4. Error codes: 404 if there is no gate to recover information from

## 2.3 UserData (Port: 6000)

### 2.3.1 - "/users/code"

1. Method(s): POST
2. Input data: istNumber
3. Output data: Returns a JSON with the new user secret for the user istNumber
4. Error codes: 400 - if the request is not well formatted

1. Method(s): GET
2. Input data: istNumber
3. Output data: Returns a JSON with the user secret and token of the istNumber's user.
4. Error codes: 400 - if the request is not well formatted

### 2.3.2 - "/users/checkcode"

1. Method(s): GET
2. Input data: A determied user's user secret
3. Output data: Returns a JSON with the user's istNumber and code equal to 0 (error) or 1 (valid)
4. Error codes: 400 - if the request is not well formatted

### 2.3.3 - "/user/id/stats"

1. Method(s): POST
2. Input data: istNumber, date , gateid
3. Output data: Creates a new UserStats instance based on the information of the input
4. Error codes: 400 - if the request is not well formatted

1. Method(s): GET

2. Input data: istNumber

3. Output data: Returns a JSON with a determined user stats

#### 2.3.4 - "/oneuser"

1. Method(s): GET

2. Input data: Username , istNumber and token

3. Output data: Returns a JSON with the userid if it works well, returns a JSON with userid = -1 if not.

#### 2.3.5 - "/users"

1. Method(s): POST

2. Input data: userName, istNumber, token

3. Output data: Creates a brand new user using the information above

4. Error codes: 400 - if the the input is not well formatted

1. Method(s): GET

2. Input data: Nothing.

3. Output data: Lists all users in a JSON

# 3 Data models

Our Data Model for this final project was a little more complex than the first one but it still looks pretty simple. This was the part that we did first because it is one of the most important ones.

Before starting a project, in a company or a personal one, we always need to think how the data model should be and how many tables, columns and relations they should have. When we were brainstorming about this final project we realized that we needed at least 2 tables, one for the users that are going to be entering the gates, and one for the gates that are going to be created in our web application. It is important to say that the users and gates database are separate. To complement this 2 tables and to save the statistics of each user and gate we realized that we also needed to create 2 more tables, one with the user stats and another with the gate stats. The data model that we sent in the submission can be seen in figure 1.

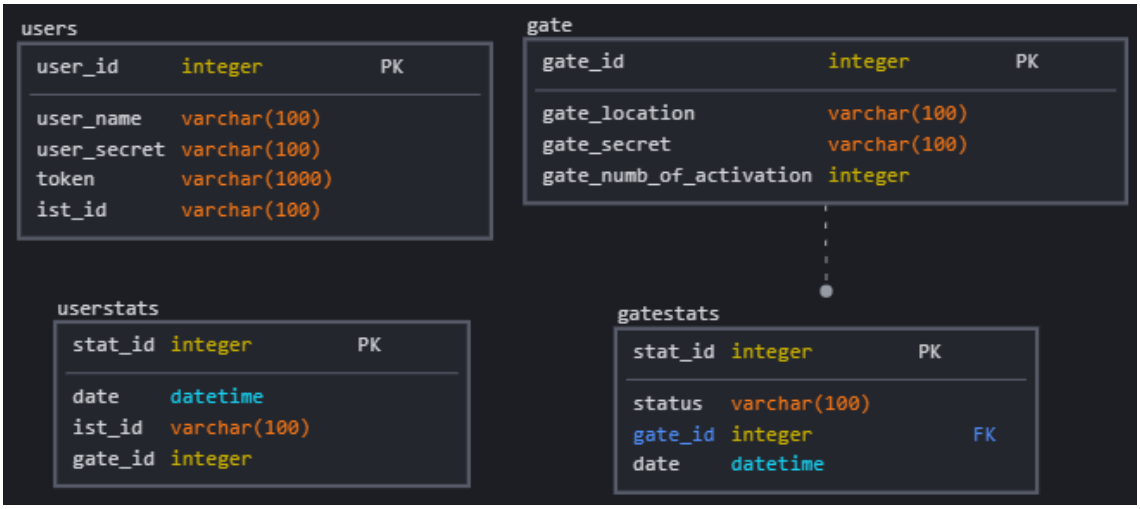



**Figura 1**

The column names are very self explanatory so there is no need to explain one by one the meaning of them. After submitting the code in fenix we realized that there were some changes that we could have done to the data model. Not because it would change the way the application works but because we were not using the standards that are used in relational databases .

For example, if right now we could re-submit the project we would change our data model to the one in figure 2.

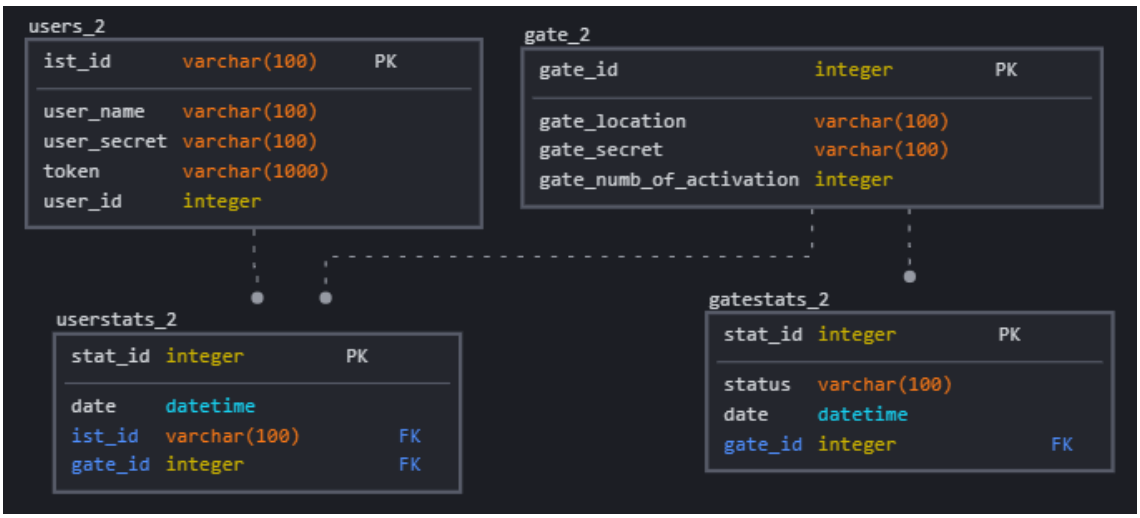



**Figura 2**

We would do this way if we could because is going according with the standards of relational databases. It is done this way because if we work with a lot of data (not the case) it would be much easier and faster to access the data we want. All we needed was to do a query, for example, searching for the number of distinct users that tried to access the gate with gateid = 3.

# 4 Functionalities

List of functionalities:

- 1. login
- 2. User request for a new code
- 3. User access to the list of accesses
- 4. Code validation by the gate
- 5. Administrator gate creation
- 6. Administrator listing of gate operations

## 4.1 Login

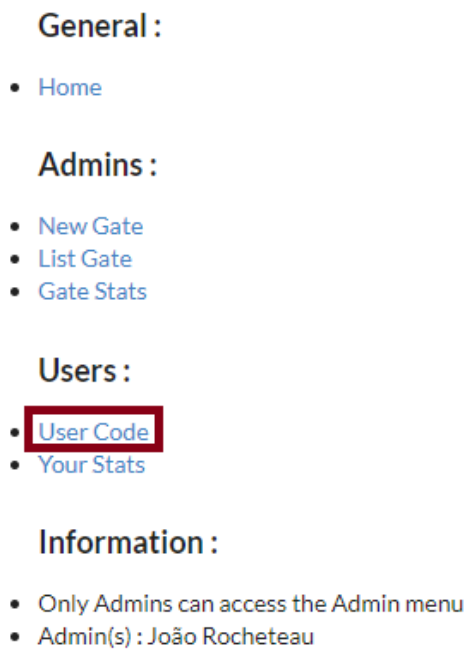
The login is processed using the OAuth library. The login starts in the Service's "/"endpoint, where a OAuth session is created and the user is redirected to IST Fenix, given the provider's authorization base url, the client ID (from the IST fenix application created) and the redirect URI. The oauth state is stored in a session variable with the name "oauthstate". The quality of the input is confirmed by IST Fenix.

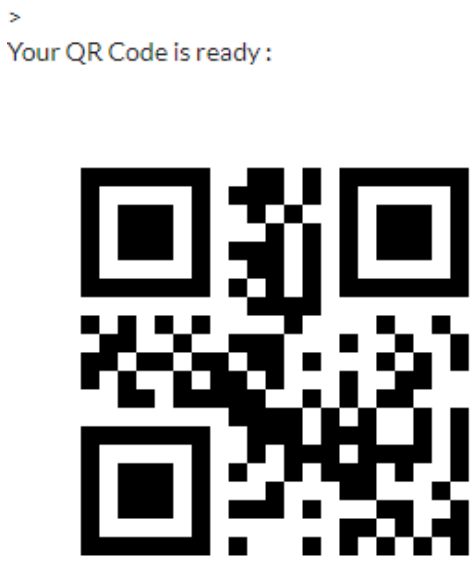

After the login is made, the user is redirected to Service's "GET /callback"endpoint where it will be given an access token that will be stored in a session variable "oauth token", this will be important to verify the token through the application. The callback redirects the user to the final step of the login, the "GET /authsuccess". The authsucess endpoint consists in working with the OAuth's supplied user information. It calls the UserData's endpoint POST /users endpoint to create a new user in the UserData database in case it's the first login (confirmed depeding of the

return to UserData's GET /oneuser, that answers with a json with a variable named "code". If the result is -1 , the user is not in the database) . It also stores the istNumber or istid, given by the first initial letters ist(and XXXXXXX numbers) on a session variable so it's possible to recover the istNumber in any endpoint and therefore allow different users to use the application.

## 4.2 User request for a new code

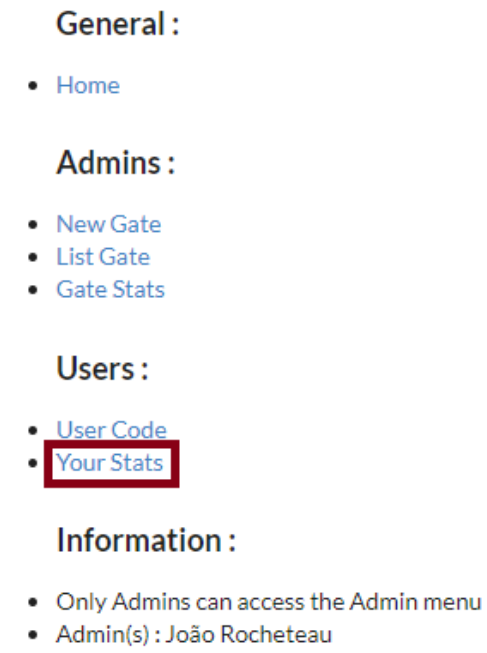
The user request for a new code is simply done by accessing Service's "GET /gates/newqrcode"where it will created (or shall we say, updated) a new user secret in the UserData database (using the database's "POST /user/code"endpoint ) and therefore a new QRCode aswell. This was made by sending a JSON with the istNumber attached to it and with the user secret returned as an output. This code is displayed, as an image, using the "qrcode"python library in which it creates a brand new qr code for each user secret. The QR Code is stored in the /static folder and is updated every time a user request a new code. Knowing where the QR Code image is stored it's now sent to the 'giveQR.html' web page and displayed there. In the User Application the user must click in User Code (GET /gates/newqrcode) to activate this procedure.






## 4.3 User access to the list of accesses

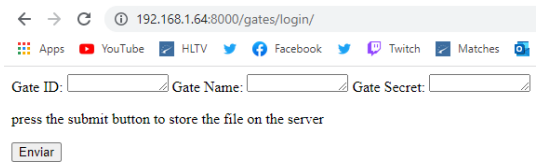
Another functionality that was requested was that the user could see, on the UserApp, all the accesses he/she was making during the application life time. As soon as the gate is accessed by any user it's known that the endpoint in which the users will be redirected is Service's "/gates/gateid/user/userid/confirmed", and , with that in mind, we made every update and/or creation that needed an access confirmation in this endpoint. One of the creations we made is introducing a new row in UserStats with the gate in which the user accessed and the date it happened. This is made creating a JSON that brings this information to another endpoint in the Service code, 'POST /user/userid/stats' that will therefore call UserData and create it( using UserData's POST /user/id/stats with the information before wrote). If the user wants to access his stats, he/she must click on the Your Stats hyperlink in which he/she will be redirected to Service's "user/stats"endpoint, a simple GET endpoint that renders the information that is present in UserData's UserStats of the user is question. This information is resolved accessing UserData's "GET /user/stats"with a get request with a JSON attached with the istNumber (from the session, making it ready for different users to use the app at the same time) that will be used for the query.




| STAT ID | IST ID | GATE ID | DATE |
|---|---|---|---|
| 1 | ist1102247 | 1 | 14/11/2021 10:44:51 |
| 2 | ist1102247 | 1 | 14/11/2021 10:45:38 |
| 3 | ist1102247 | 2 | 14/11/2021 10:49:13 |
| 4 | ist1102247 | 2 | 14/11/2021 10:50:47 |
| 5 | ist1102247 | 2 | 14/11/2021 10:50:52 |
| 10 | ist1102247 | 1 | 14/11/2021 11:01:17 |
| 11 | ist1102247 | 1 | 14/11/2021 11:02:23 |

## 4.4 Code validation by the gate

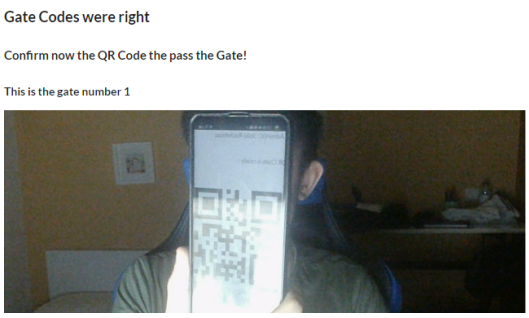
Before the user inserting the QR Code, it's needed to make a gate validation interface. The User, after inserting the gate location , gate id and secret, (the gate login is done in GET /gates/login, where this simple interface will show up) is redirected (coded in the html page) to the Service's "POST /gates/login/confirm"endpoint in which all the verifications are made.



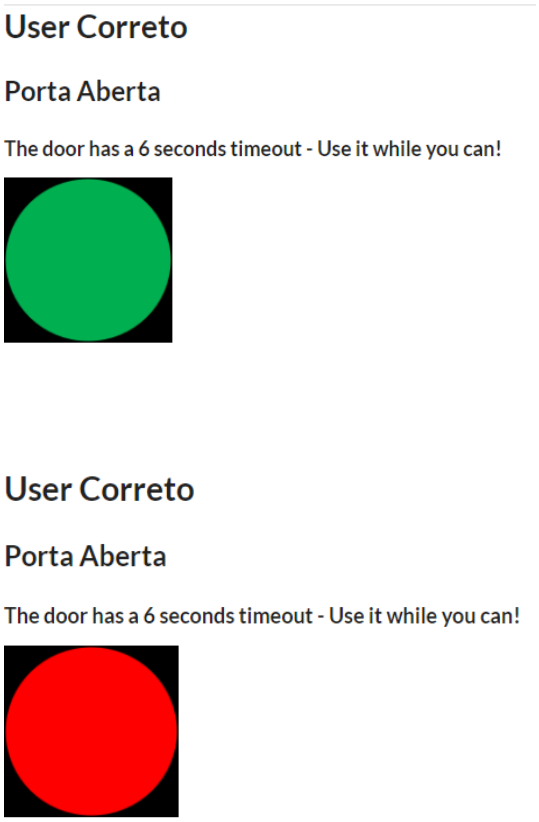

Here, it's confirmed if the gate secret has no spaces and if the gate location and gatesecret are a string. After this, the request is in a valid format. To the check if the gate id and gate secret

are just fine, we make a get request to GateData with a JSON with the gate's id on the endpoint ("/gates/code") that will return a json with the gate secret that is currently in the database. Using a simple if we confirm that the gate secret is or not good for that specific id. We present a web page in which the user will be able to show his QR Code (ScanQR.html) in case this tests are passed. For the User Code (QR Code) confirmation, it's imperative the use of Java Script.

First of, ScanQR will activate a determined gate camera, in which, using instaScan library, will capture the input string code of the user everytime he presents a QR code in front of it. Having this code, its needed now to confirm if the code is in line of what the user should have and , to confirm, the java script code will call the "/gates/id/checkcode"endpoint in the service using the AJAX library with a POST request. This endpoint will work directly with the UserData database, making a get request with a JSON that features that same input user code to the endpoint 'GET /users/checkcode'. This UserData's endpoint either a JSON with code = -1 (in case it fails, the User Code is not the same as the database's code ) or a JSON with code =1 (in case the codes are the same).



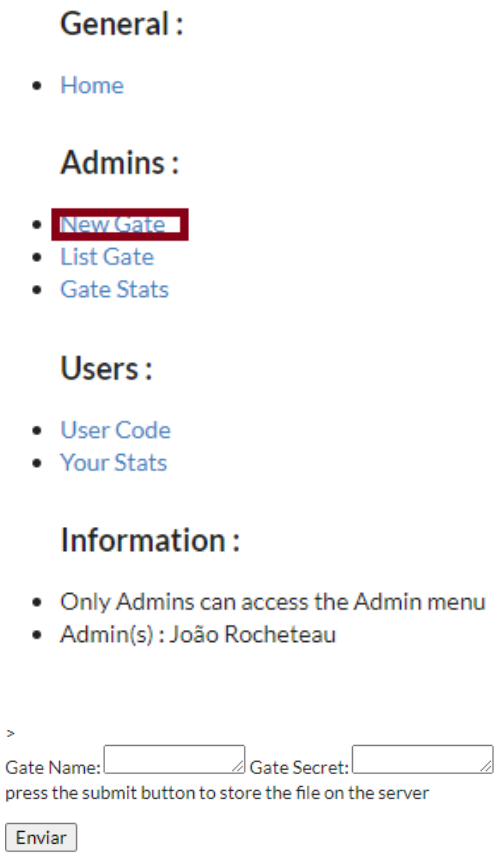

Making use of the UserData's response , this Service endpoint returns either a JSON with id = -1 (in case it fails, the User Code is not the same as the database's code ) or a JSON with id = istNumber (in case the codes are the same). This returns are presented to the AJAX code and are useful to understand if the gate authentication was successful or not. Now all it rests to do, is for the AJAX code to decide to which endpoint the next step is. If the code is right it will go to Service's "GET gates/id/user/id/confirmed", if it's not, "GET gates/id/user/notconfirmed". "Confirmed"endpoint will render the templates of the gate being opened or not using a gif that represents a green and red light (usercodegood.html), while the "Not Confirmed"will show an error message (userbadgood.html) For the green /red light gif, we used a simple JQuery's hide() and show() mechanism that made the green light be active for 6 seconds.



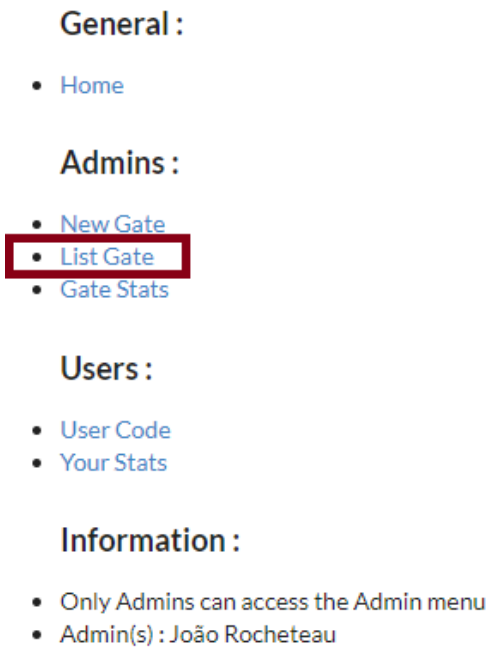

## 4.5 Administrator gate creation and Administrator listing of gate operations

To make a Admin-only-operation, for both endpoints, all it was needed was a simple if that verifies if the istNumber is from an admin or not (using the current session istNumber). They were created at the start of Service's file "/gates"endpoint and another in the /gatestats.

/gates is an endpoint that is present in both GateData.py and Service.py files, where it's objective is to create a gate, in case the request is a POST or to list all the gates if the request is a GET. This point is direclty used in the UI and by making it unreachable to any normal user confirms that the gate creation is only made by admins (the normal users, when trying to create a gate will be redirected to the menu again). When doing a POST request in Service's "/gates", another POST will be made to the GateData's "/gates"featuring a JSON message that will have the information the admin introduced in the gate creation form. When doing a GET request in Service's "/gates", another GET will be made to the GateData's "/gates"needing no input data, just a query that represents all gates.




Listing all the gates is simple too:



| ID | Name | Secret | Numb of activ |
|---|---|---|---|
| 1 | Local | Local | 7 |
| 2 | 1 | 1 | 5 |

/gatesstats is an endpoint that is present in both GateData.py and Service.py files, where it's objective is to create a gate, in case the request is a POST or to list all the gates if the request is a GET. This point is direclty used in the UI and by making it unreachable to any normal user confirms that the gate creation is only made by admins (the normal users, when trying to create a gate will be redirected to the menu again). When doing a POST request in Service's "/gatestats", another POST (creation) will be made to the GateData's "/gatestats"featuring a JSON message that will have the information recieved by the Service's INPUT JSON with status (if the user could pass or not the gate) , the date it happened and the gate used. When doing a GET request in Service's "/gatestats", another GET will be made to the GateData's "/gatestats"needing no input data, just a query that represents all gatestats.

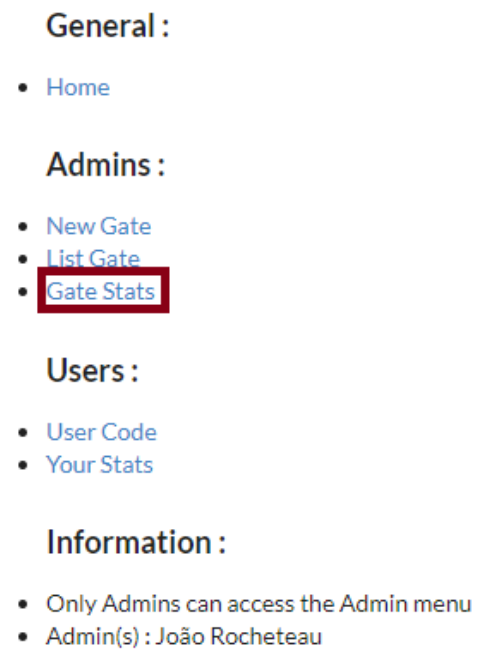



**Figura 3**

| Stat ID | Status | Gate ID | Date |
|---|---|---|---|
| 1 | Failed | 1 | 14/11/2021 10:43:17 |
| 2 | Success | 1 | 14/11/2021 10:44:51 |
| 3 | Failed | 1 | 14/11/2021 10:45:01 |
| 4 | Success | 1 | 14/11/2021 10:45:38 |
| 5 | Failed | 2 | 14/11/2021 10:46:21 |
| 6 | Failed | 1 | 14/11/2021 10:47:04 |
| 7 | Failed | 2 | 14/11/2021 10:48:19 |
| 8 | Failed | 2 | 14/11/2021 10:48:45 |

# 5 Validations

List of Validations :

- 1. verification that a user is logged in
- 2. verification of the gate id/secret
- 3.verification of the QRCode
- 4. timeout of the QRCode
- 5. validation of the requests input
- 6. verification of the requests error codes

## 5.1 verification that a user is logged in

Already explained in the last chapter.

## 5.2 verification of the gate id/secret

Already explained in the last chapter.

## 5.3 verification of the QRCode

Already explained in the last chapter.

## 5.4 timeout of the QRCode

To create a the timeout of the QR code we found a way of having a timer actively counting the seconds since the user generated a new QRCode and change it as soon as 30 seconds passed. In service's endpoint GET /gates/newqrcode, where we created a new code to be displayed, we also created a thread that, as soon as the 30 seconds pass, it will go trough a function called "timerfunction()"that will change the User Secret and therefore making the old one invalid. The timerfunction() has a istNumber as an input variable, creates a JSON with it, and creates a POST request in '/users/code' with the same istNumber, updating the code of the determined user.

## 5.5 verification of the requests input

The verification of the input data was made at the start of each function using an if.

## 5.6 Input Verifications / Verification of the requests error codes

### 5.6.1 Service Input Verification

The two main service's input verification made were:

1. if(str(session['oauth token']) == requests.get(ipuserdata + '/users/code', json = 'ist id' : session["istNumber"]).json()['token']) :

2. if(session["istNumber"] == 'ist1102247' ) :

1.Checks if the authetication token is valid by seeing if the token in the database is the same.

2.Checks if the current session user is an admin - will allow if it is.

In case a number 1 verification falls short, the user will be redirected to a error webpage(invalidtoken.html). In case a number 2 verification falls short, the user will be redirected to a error to the menu again, having no acess to that endpoint (menu.html).

Endpoints that used the nº1 verification :

Number 1 verification :

- "GET /menu"
- "GET /newgate"
- "GET /gates/newqrcode"
- "GET /gatestats"
- "GET /user/stats"
- "POST,GET /gates"

Number 2 verification :

- "GET /newgate"
- "GET /gatestats"
- "GET,POST /gates"

We still do two more verifications :

- "GET,POST /gates"

if((isinstance(newgatelocation, str)) and (isinstance(newgatesecret , str)) and ((' ' in newgatesecret ) == False))

Here, it's confirmed if the data that was introduced in the /newgate form had no flaws. If it has, returns a template which shows that it was made an error (render$_t emplate('gatecodebad.html')$)

- "POST /gates/login/confirm"

if((isinstance(newgatelocation, str)) and (isinstance(newgatesecret , str)) and ((' ' in new gatesecret ) == False))

Here, it's confirmed if the data that was introduced in the /gates/login form had no flaws. If it has , returns an abort(400) - Bad request

### 5.6.2 GateData and GateDataReplica Input Verification

- Every Database creation has a commit() rollback() system.

- In most cases, whenever a query is made, a try expect is used in order to catch an error in case there is no information to be queried and delivers an abort (404) - Not Found

- "/gates/code"

if ((isinstance(request.json['id'], int) or isinstance(request.json['id'], str)) and ((' ' in request.json['id']) == False)) :

Confirms if the id received in a input json is an integer or string and checks for any spaces. If it fails returns an abort(400) - Bad request

- "/gatestats"

if ((isinstance(request.json['status'], int) and isinstance(request.json['date'], str) and isinstance (request.json['gateid'], str)) ):

Confirms if the status is an integer, if the date is a string and the gate id is a string. If it fails returns an abort(400) - Bad request

- "/gates

if((isinstance(request.json['gatelocation'], str)) and (isinstance(request.json['gatesecret'], str)) and ((' ' in request.json['gatesecret']) == False)) :

Confirms if the gate location is a string, if the gate secret is a string and if it has any spaces. If it fails returns an abort(400) - Bad request

### 5.6.3 UserData Input Verification

- Every Database creation has a commit() rollback() system.

- In most cases, whenever a query is made, a try expect is used in order to catch an error in case there is no information to be queried and delivers an abort (404) - Not Found

- "POST , GET /users/code"

if ((isinstance(request.json['istid'], str)) and ((' ' in request.json['istid']) == False)) :

Confirms if the istNumber recieved has no spaces and is a string. If it fails returns an abort(400) - Bad request

- "GET, /users/checkcode"

if ((len(request.json['code']) == 6) and ((' ' in request.json['code']) == False )) :

Confirms if the code received has 6 digits like all the others and that there are no spaces. If it fails returns an abort(400) - Bad request

- "POST /user/id/stats"

if ((isinstance(request.json['istid'], str) and isinstance(request.json['date'], str) and isinstance(request.json['gate
str) and ((' ' in request.json['istid']) == False)) ):

Confirms if the istNumber recieved, the date and the gate id are a string. Also check if there are any spaces in the id. If it fails returns an abort(400) - Bad request

- "POST /users"

if ((isinstance(request.json['username'], str) and isinstance(request.json['istid'], str) and isinstance(request.json
str) and ((' ' in request.json['istid']) == False)) ):

Confirms if the user name, the istNUmber and token are string. Checks if the id has any spaces. If it fails returns an abort(400) - Bad request

# 6 Fault tolerance

It was asked to create a replica of the GateData service so we could have a backup if the database was down our if it was just impossible to get or post the data from or to there.

To do this, it was created an exact duplicate of GateData.py where the only thing that changed was the port. The port for GateData.py is 7000 and for the GateDataReplica.py is 9000. Both of the services need to be running at the same time so it is possible to post, get or put the records in both of them. In the GateData service was the gate database where we stored the gate and gatestats table. When we duplicated the service we also duplicated the database, and the structure stayed like we can see in figure 4.

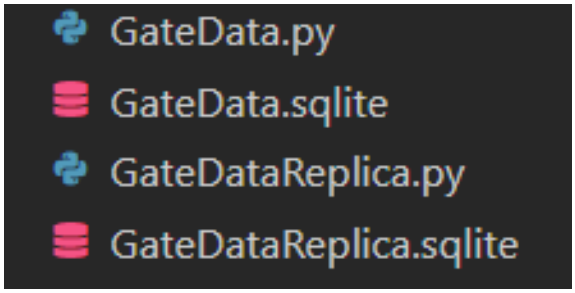



**Figura 4**

To use this technique correctly all we needed to do was to duplicate the requests we did on Service.py. Everytime the application sent a request.post to the GateData service we also sent one to the GateDataReplica service. This way we always had both of them updated and with the same information.

To get the information we had to create a method where if the request.get failed in the first database (GateData.sqlite) we needed to go to the backup to get the data we needed. To do this we implemented the try and except method directly in Service.py, like it can be seen in figure 5.

```
try:
    response = requests.get(ipgatedata + '/gates')
except:
    response = requests.get(ipgatedatareplica + '/gates')
```

**Figura 5**

# 7 Used libraries

## 7.1 Python :

- Flask : Flask, jsonify, request , abort , render template , session, url for , redirect
- SQLAlchemy : create engine , declarative base , Column, Integer , String , Foreign Key , relatioship , session maker
- String
- Random
- OS : error
- Requests
- DateTime
- qrcode
- requests oauthlib : OAuth2Session
- threading

## 7.2 Java Script :

- jQuery
  - Fomantic UI
  - AJAX
- InstaScan

# 8 Project Changes

It is safe to say that almost everything changed from this final project to the intermediate one.

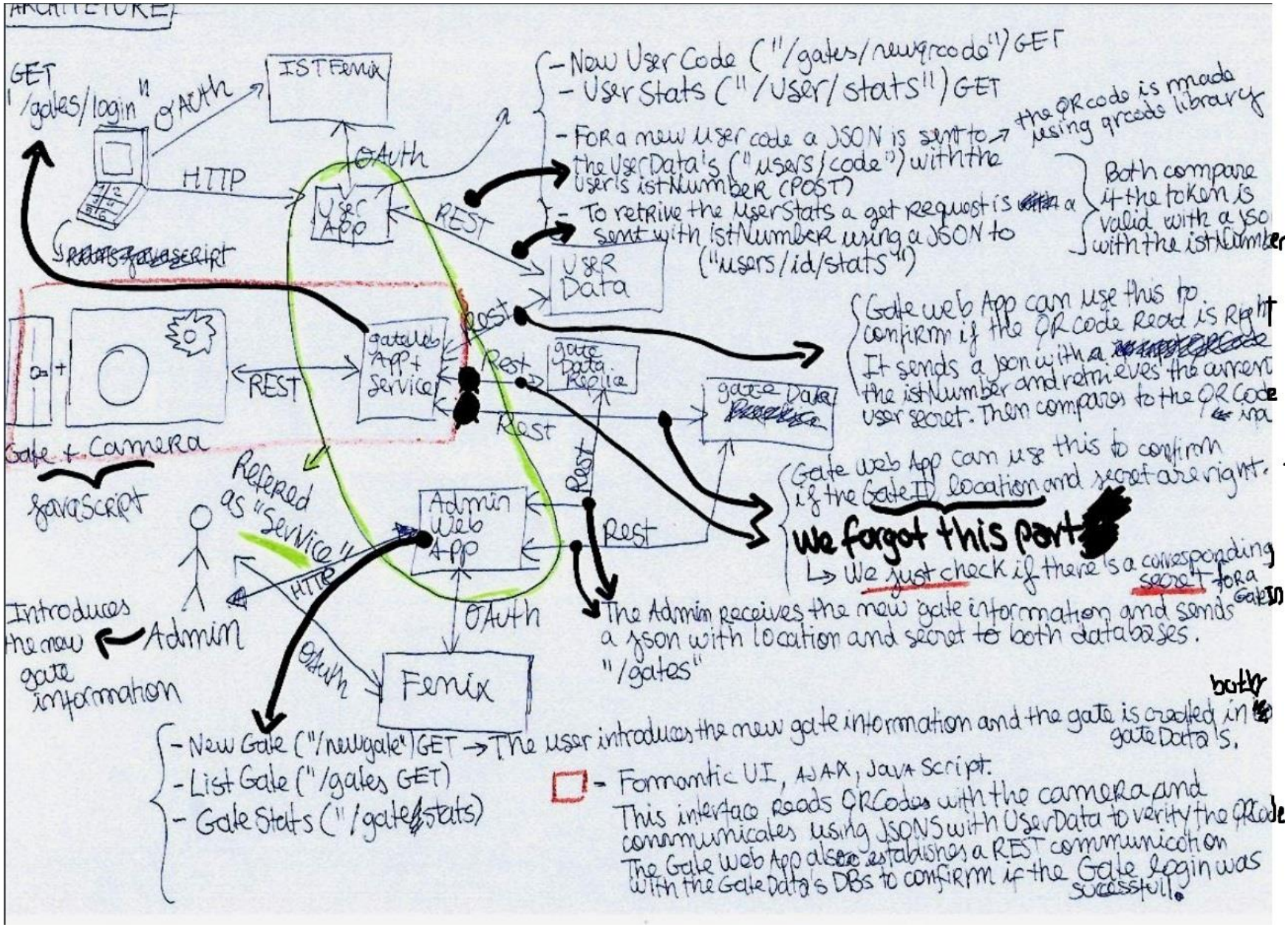
Starting from the Architecture, in the first version of the project we were using UserApp and GateApp to change the user secret and to enter a gate, respectively. In the final version we had not only a section in our UI where the user could even authenticate with the IST credentials, but also a place where he/she could enter a gate and see those statistics. We added a new feature that was that the users didn't needed to do complete a forms to create a user in the database.What we did was create a new user every time a new one authenticated in the application.

The data model also changed. We added 2 more tables and separated the two databases. It was also added a backup database for the gates, more known as GateDataReplica.

We also added some functionalities starting by the new UI. Right now is possible to use the application only via the web page and not anymore from the terminal. We stopped using the UserApp and GateApp and started to have the QR Code generator and validator all integrated in our UI. It were also added some conditions to warn the user if the QR Code was the correct one or not.

The Administrator privileges was also a new feature. Some of the tabs in the application can only be accessed by the Administrator, for example the list of gates and the tab where it is possible to create a new gate.

The last functionality that we added was that the user authenticated in the application was the only one that could see his information in the "User Stats"tabs. This means that if another user authenticates in the application, it would not be able to see the other user information.


ARCHITETURE
GET "/gates/login"
OAuth
IST Fenix
HTTP
OAuth
User App
REST
- New User Code ("/gates/newqrcode") GET
- User Stats ("/user/stats") GET
- For a new user code a JSON is sent to the UserData's ("users/code") with the User's istNumber (POST)
- To retrive the UserStats a get request is sent with istNumber using a JSON to ("users/id/stats")
the QR code is made using qrcode library
Both compare if the token is valid with a JSON with the istNumber
User Data
Gate web App can use this to confirm if the QR code Read is Right
It sends a json with a
the istNumber and retrieves the current user secret. Then compares to the QR Code
Gate + Camera
JavaScript
gateUser App + Service
REST
Post
Rest
gate Data Replica
gate Data
Rest
Rest
Rest
Referred as "Service"
Admin Web App
Gate web App can use this to confirm if the Gate ID location and secret are right.
We forgot this part
We just check if there is a corresponding secret for a Gate ID
HTTP
OAuth
Introduces the new gate information
Admin
OAuth
Fenix
The Admin receives the new gate information and sends a json with location and secret to both databases.
"/gates"
both
- New Gate ("/newgate") GET → The user introduces the new gate information and the gate is created in gate Data's.
- List Gate ("/gates GET)
- Gate Stats ("/gate/stats)
- Fomantic UI, AJAX, Java Script.
This interface reads QRCodes with the camera and communicates using JSONS with UserData to verify the QRcode
The Gate web App also establishes a REST communication with the Gate Data's DBs to confirm if the Gate login was successful.