Internship Report

Mattia Evangelisti
s268637

June 2022

An internship report presented for the degree of
Computer Engineering

Contents

1 Summary

At the beginning of the academic year I chose to substitute my 3^{rd} free ECTS exam with an internship. For my study programme was slated a 10 credits internship. Since I decided to work part-time for the whole experience, it last 3 months.
During my internship experience with Ubroker, I have developed my own data mining project from scratch, developing the necessaries skills and knowledge.
The main objective of the project was to retrieve data about employees, company devices and their usage, and to create reports that would be useful for the company.
The project was developed in Python, language that I learned thanks to the constant support of my colleagues.
The key points of the project could be summarized as follows: data retrieval from Microsoft Azure, through the use of Microsoft Graph APIs, data manipulation, data storage in an internal database, and the creation of reports.
Although I found the project to be a challenging experience, I found it to be valuable in developing my skills in Python programming, data manipulation and database management.

2 Company

Ubroker srl was founded in 2015 with the goal of providing eco-friendly energy to their clients, without neglecting the need of a competitive price.
More information can be found in their website.

2.1 Mission

The company's mission is to improve the life quality of its clients by granting them a simple management system that allows them to control their energy consumption and a cost saving service, providing competitive prices below the market average. Moreover, Ubroker is constantly striving to maintain the social responsibility by choosing providers who follow policies that safeguard the environment and cultures involved in the energy distribution process.

2.2 Market Strategy

Ubroker is an electricity and gas provider, that collocate itself at the last link of the energy distribution chain. Their working area is indeed the sales to the final client.
The company sales target is the domestic market, the majority of their customers are private citizens. The most recent data report 85,000 active clients, who subscribed 52,000 electricity contracts and 32,000 gas contracts.
The peculiarity of Ubroker's market strategy is the system called *scelgo zero*. Each client can join the zero project though their platform and start to reduce to zero their electricity and gas bill.
On their website, each client may have two options: to become a testimonial and start to zero their bills and the possibility to become an advisor of the zero project. The first option allows the clients to reduce their bill, indeed each user will receive points every time they invite a new client and every time an invited user bring in someone else. Through this mechanism, the bill will be proportionally reduced, possibly until zero. On the other hand, by choosing the second options, a user may become an external advisor, whose role will be to sell contracts to new potential clients, earning a commission for each new subscription. Moreover, those external partners will have access to a wide range of training courses, that can be chosen and followed on the platform.

2.3 Economic Growth

Ubroker was founded in 2015 and since then, thanks to the innovative market strategy and the strong digitalization of all the services, the company has been able to grow rapidly. Thanks to those characteristics, they were able to achieve incredible goals, such as a total of 170000 clients and over 2 million issued invoices. Moreover, the company has recently gained a Microsoft partnership.

2.4 Working environment

The company currently employs about 50 people, divided in various departments, such as sales, marketing, IT, legal, accounting, etc.
The main office is located in the city of Collegno, in the metropolitan area of Turin and, at the moment is composed of one building, but they will be able to expand their space in a second build, within the next few months. Moreover, the company is still granting to the employees the possibility of working remotely, through the use of devices provided by the company itself.

2.5 IT department

Being a computer engineering student, I spent most of my internship in the IT department, which is composed of a team of 8 people. The main goal of this department is the software development, for both internal and external use.
The team covers all the aspects of a software life, from the development of the software itself, including both front-end and back-end development, to the integration and the maintenance of the software with the company's infrastructure.
The main projects of the IT department are: *Piattaforma Zero*, which was described before, several web applications used by the other departments and an internal system for the management of the company's contracts.
All the software are developed using the Docker technology, which allows to create a containerized environment, in which the software can be developed, installed and run. The front-end developing is mostly done using the JavaScript framework Angular, on the other hand, the back-end developers deals with PHP framework Laminas and PostgreSQL databases.
Moreover, I found fascinating how the whole infrastructure is cloud based. The company make indeed an intense use the Azure cloud, which is a cloud computing platform owned by Microsoft, that allows to have storage and virtual machines in the cloud. Ubroker uses about 60 virtual machines and

over 6 TB of storage on the Azure platform. Thanks to this intense use of this service, the company is recently become a Microsoft partner.
My role in the department was to create a software able to retrieve data about employees, company devices and their usage, and to create reports that would be useful for the company.

3 Project Introduction

The project can be summarized in 4 main parts:

- Data retrieval from Microsoft Azure, through the use of Microsoft Graph APIs.
- Data cleaning and manipulation.
- Data storage in an internal database.
- Report creation using Power Bi.

3.1 Used technologies

The project was entirely written in Python 3.9 programming language and the following libraries were used:

```
colorama==0.4.4
jsonschema==4.4.0
msal==1.17.0
pandas==1.4.2
psycopg2==2.9.3
requests==2.27.1
tabulate==0.8.9
```

For the database management it was chosen to use PostgreSQL, as long with BDeaver, which is a SQL client software application and a database administration tool.
The project was developed using the GIT version control system, integrated with the hosting system GitHub. These two tools allow to store, modify and share the project's code, as well as to keep a records of the activities and modifications done.
The data retrieving was carried out thanks to the Microsoft Graph API, which is a REST API that allows to retrieve data from Microsoft Azure.
Finally, the report creation was done using Power Bi, a Microsoft tool that allows to create reports in a simple and intuitive way.

3.2 Standards

The whole Python coding part of the project was written following the PEP8 Style Guide for Python Code. This guide gives coding conventions for the Python code comprising the standard library in the main Python distribution.
The application was developed from scratch, so I had to organize the files in order to maintain a precise schema. In particular mine was a command-line application composed by a main file, called `console.py` and several other internal packages. Following the standard layout, I divided all my files into subfolders, each one containing packages divided with respect to their use and their functionality.
Throughout the developing of my project, I made an extensive use of the Docker technology, which allowed me to create a containerized environment, isolated from the rest of the system, in which the software was developed, tested and run.

3.3 Training

During my internship experience I was able to learn several new technologies, I indeed spent the first week of my internship learning everything I would have needed for the developing of the project.
First I had to learn Python programming language, to which I was totally new. This task was carried out following different tutorials and thanks to the constant support of company tutor, who assisted me in my training. The learning of this language was not very difficult, since, thanks to the courses I followed, I already had a strong coding background.
Then I start to discover the API world, particularly I had to learn how to make REST HTTP request to an endpoint in Python. Fortunately, I had already studied the basics of the HTTP protocol and the REST architecture during the course of *Introduction to databases*, so I only had to adapt my previous knowledge to the Python language.
Finally yet importantly, I learned, how to manage my progresses using the version control system GIT. It was the first time I used it in a working environment, but thanks to the course of *Object oriented programming*, I already had experience with the SVN version control system.

4 Project

4.1 Console menu

The main file and the only one which was effectively run, is the `console.py` file. This file is composed by a main menu, which allows to choose the desired command.
The menu allows the user to choose which command to run or to open a help menu, which provide the user with information about the available commands and their usage. The selected command has to be passed as a parameter when the script is run, some commands may accept optional parameters as indicated in the help menu. If no command is specified, the default action is to open the help menu.

```
USAGE:
command [arguments]
COMMANDS:
help                    List all available commands
db-test                 Test database connection
db-mig-status           Print the status of migrations in the file system and in the database
db-mig-execute          Execute the migrations
ping                    Ping a fixed API and check the correctness of the parameter pong if passed
graph-devices           Retrive the devices from Graph API, validate and insert them in the db
graph-users             Retrive the users from Graph API, validate and insert them in the db
graph-logs              Retrive the logs from Graph API, validate and insert them in the db
stats-unlogged-users    List the users who have never done a log in a registered device
stats-unused-devices    List the devices that have never received a log in a specified timeframe
stats-shared-devices    List the logs done from different users on the devices used from more than one user
stats-users-unregdev    List the users who have done logs only on unregistered devices
stats-far-logs          List the logs done in far places in a short time interval
```

Figure 1: help menu

Summing up, the main role of the console is to import all the required packages and execute the commands, through all the function contained in the other files.

4.2 Configuration

The parameters needed in the project, such as database information, API URLs, JSON files and other data files, are stored in the `config.ini` file. This allows the user to have a much more clear overview of the possibility to change the information based on the current needs.
On the other hand this data are not directly contained in the Python code,

so they need to be retrieved. The ConfigParser package comes in our help, allowing as to retrieve the data from the config file and return them in any Python function.

```
def get_url():
    parser = ConfigParser()
    parser.read("config.ini")
    url = parser.get('ping', 'url')
    return url
```

The above code is an example of how the configuration function works. In the example, the parameter url is retrieved from the section ping of the config.ini file.

4.3 Database Connection

The first important step of the project was to set up a database. As said in the previous sections, the database was a PostgreSQL database, hosted in the Azure cloud.
One the database was up and running, the next step was to create a connection between the Python application and the database, in order to be able to store, modify and delete data directly from the application.
It twas chosen to use the Psycopg2 library, which is the most popular PostgreSQL database adapter for the Python programming language. This package allows to open a connection, query the data, commit the changes and finally close the connection.

```
# parameters are host, database, user, password
def connect(params):
    conn = None
    # connect to the PostgreSQL server
    conn = psycopg2.connect(**params)
    # create a cursor
    cur = conn.cursor()
    # execute a statement
    cur.execute('SELECT version()')
    # display the PostgreSQL database server version
    db_version = cur.fetchone()
    print(db_version)
    # close the communication with the PostgreSQL
    cur.close()
    return conn
```

The above code describe the connect function, that is one of the most fundamental functions of the project. It opens a connection with the database, then the cursor object allows to execute a query and retrieve the result.

Finally, the cursor is closed and the `conn` object is returned, in order to be used wherever is needed a database connection.

4.4 Migrations

During the development of the project the database was constantly modified: some tables was added, removed or altered. In order to keep track of all the changes, it was created a table named `migrations`, used to store the changes made to the database. The table is composed of two columns: the migration version and exact date and time of the entry insertion.
The concept of migrations was widely known in the company IT department, where the team makes a intense use of the migration doctrine, applied to PHP. In order to aligne mt project with the company standards, it was chosen to use the concept of migrations.
There exist Python libraries useful to manage all the migrations, but since this is a small project, instead of learn the use of a complex library, it was chosen to personally develop the functions needed for the management of the database migrations.

	version	apply_at
1	202203211221	2022-03-21 12:21:17.832 +0100
2	202203280945	2022-03-28 09:55:09.600 +0200
3	202204011155	2022-04-01 12:02:23.569 +0200
4	202204051040	2022-04-05 10:52:11.396 +0200
5	202204081055	2022-04-08 11:05:16.294 +0200
6	202204081205	2022-04-08 12:41:33.462 +0200
7	202204111040	2022-04-11 10:55:58.101 +0200

Figure 2: table migrations

Each time it was needed to modify the database, a SQL file was written and named with current date and time. Then it was checked if the instruction written into the file were ready to be applied to the database.
First it was checked if the name of the migration file respects the naming constraints, i.e., thanks to the `datetime` library, it was checked if the name was a valid date and time format. The next step was to find out if the file was not already present in the database and if it was the most recent. If both conditions were verified, then the file was inserted in the *ready to be executed* list, meaning that the migrations was ready to be applied to the database.

The main menu offers two migrations command:

- `db-mig-status`: allows the user to have an overview of the migrations' status. It lists the migrations ready to be executed, the ones available only in the file system and those available only in the database. The last two are indicated as errors, since the file systems and the database migrations table must be aligned.

- `db-mig-execute`: execute all the migrations labelled as *ready to be executed*. If the execution is successful the changes are committed to the database and a new entry is inserted in the migrations table. If the execution is not successful, the changes are rolled back and the user is informed.

4.5 Azure Application

4.5.1 Azure Environment

Azure is a cloud computing based platform for hosting applications and website, created and operated by Microsoft. After registering an application in the Azure portal, one can access and use all the services provided by Microsoft Graph.

4.5.2 Application Registration

The project objective is to retrieve and analyze the data from Microsoft Graph. In order to be able to use this service, I had to register my application in the Azure portal.
Following the official documentation and thanks to my company tutor support, the application was correctly registered and the required permission was given. After the registration procedure, the application details were generated by the portal: client ID, secret, authority and scope. This data was saved in the `config.ini` file since they would be used soon to access the Microsoft Graph API.

4.6 Microsoft Graph

4.6.1 Authorization and Connection

Once the application was registered in Azure platform, in order to access the Microsoft Graph API, I had to connect my Python code to Microsoft Graph API.

Microsoft Azure makes use of OAuth 2.0, which is the industry-standard protocol for authorization, based on the concept of access tokens.
Authorization and connection was implemented following the official guide and using the `msal` package.
Once the connection was established, an authentication token was generated, which is later used to access the Microsoft Graph API.

```python
def connect(authority, client_id, scope, secret):
    app = msal.ConfidentialClientApplication(
          client_id, client_credential=secret,
          authority=authority)
    result = app.acquire_token_silent(scope, account=None)
    if not result:
        result = app.acquire_token_for_client(scopes=scope)
    return result['access_token']
```

The above code snippet explains how the connection is established. The needed parameters are the client authority, ID, scope and secret. Then is checked if the token is still valid. If it is not, it is renewed and at the end the token is returned and ready to be used to perform the requests to the Microsoft Graph API.

4.6.2 Graph: Users

The first set of data which I worked with is the list of all the users registered with a company Microsoft account.

Data Retrieval

The first step was to retrieve the list of users from the Microsoft Graph API. The task was performed, as explained before, with a GET request, through the `requests` library.
An endpoint was needed and the one I used had the following url:

`https://graph.microsoft.com/v1.0/users?$top=999&$select=id,displayName`

where the `$select` parameter is used to retrieve only the fields needed.
The request is done inserting in the header the authorization token, whose importance was explained before. Finally, if the request was successful, the list of users is returned.

Response Validation

Once the response of the endpoint was available, it needs to be validate, i.e., it must be checked that all the required fields are present and in the correct format.

The response of every API is a `json` text, so it is checked against a `json` file specifically writte for the used endpoint. This is performed using the `validate` module of the `jsonschema` library. The following snippet reports the `json` file written to check each object of response of the users endpoint:

```
{
"id": {"type": "string"},
"displayName": {"type": "string"},
"required" : ["id"]
}
```

It states that two fields are expected: `id` and `displayName` and both must be of string type, i.e., they must be a text. Moreover, the `required` instruction imposes that the `id` field must be present, othterwise the validation will fail, while the `displayname` may be absent.
This `required` instruction was specified for the `id` since it will later be used a primary key in the database, so it cannot be a null value.

Database Insertion
Finally, after the data was retrieved and validated, they are ready to be inserted in the database.
Through a migration (see section 4.4 for furhter information) the table `users` was created, with the following characteristics:

Column Name	#	Data type	Identity	Collation	Not Null
user_id	1	text		default	[v]
displayname	2	text		default	[ ]

Figure 3: table users

One the users list is available, it must be inserted in the database. This is done though a cycle and for each entry of the list there are two possiblities:

- `insert`. This command insert a user in the table if and only if the user is not already present in the table.
- `update`. If the user is already present in the table, this command updates the user's information.

FInally, at the end of the cycle, the changes to the database are committed and the connection is closed.

4.6.3 Graph Devices

The second big set of data used in the project was the list of the company devices given to the employees.

Data Retrieval

Such as for the users, first step was to retrieve the list of devices from the Microsoft Graph API. The task was performed, with a GET request, through the `requests` library.
An endpoint was needed and the one I used had the following url:

```
https://graph.microsoft.com/v1.0/devices?$top=999&$select=deviceID,
id,displayName,registrationDateTime,approximateLastSignInDateTime
```

where again the `$select` parameter is used to retrieve only the fields needed.

Data Validation

In the same way as it was explained before, the response of the endpoint was validated, i.e., checked against a `json` file specifically written for the used endpoint.
The following piece of code shows an extract of the `json` file:

```
{
...
"registrationDateTime": {"type": "DateTimeOffset"},
"approximateLastSignInDateTime": {"type": "DateTimeOffset"},
"required" : ["deviceId", "id", "displayName"]
}
```

The deviceID, id and displayname fileds are omitted since they are `string` and there are no differences with respect to what explained for the user `json` file. What is new are the `registrationDateTime` and `approximateLastSignInDateTime` fields, which are dates and are expressed as `DateTimeOffset` format.

Active Devices

The problem of the devices list retrieved from Microsoft Graph is that it contains not onlt the active devices, but also the devices that are not active anymore, such as old notebooks or pohones. In order to overcome this problem, the devices list was filtered to obtain only the devices effectively active in a time period, determined in the `config.ini` file.
After the validation the two date fields were converted in `date object`, foundamental for working with dates in Python. Then, the devices list was

filtered to obtain only the devices that are active in the deired period.
The following code explains how the list of the active devices is obtained:

```
def list_active_devices(devices, active_period):
    today = datetime.now()
    start_date = (
        today -
        timedelta(
            days=active_period,
            hours=today.hour,
            minutes=today.minute,
            seconds=today.second)).strftime("%Y-%m-%dT%H:%M:%SZ")
    active_devices = []
    for device in devices:
        last_login = device['approximateLastSignInDateTime']
        if last_login is not None and last_login >= start_date:
            active_devices.append(device)
    return active_devices
```

First the start date is found, subtracting the number of days specified in the period from the current date, thanks to the `timedelta` function. The obtained date is converted in a date object using the `strftime` function.
Then, the list of devices is cycled and only the devices with a last login more recent than the start date are added to the list of active devices, that is in the end returned.
At the end of the function, we will have a list containing only the devices with the last login done in a defined time period.

Database insertion and deletion

Finally, the list of active devices is inserted in the database. The process is similar to the one for the users.
The table `devices` was created, with the following characteristics:

Column Name	#	Data type	Identity	Collation	Not Null
device_id	1	text		default	[v]
id	2	text		default	[v]
displayname	3	text		default	[v]
registration	4	timestamp			[ ]
last_login	5	timestamp			[ ]

Figure 4: table devices

The main difference with respect to the users part is that the devices stores in the database must be deletedonce they exit the active period.

This simple piece of code solve the problem:

```
for device in devices:
    ...
    cur.execute(sql_check, (deviceID,))
    if cur.fetchone()[0] is False:
        insert(cur, deviceID, id, name, date, last_login)
    else:
        update(cur, deviceID, last_login)
delete_devices(cur, active_period)
...
```

It checks, for each device in the list, if it is already present in the database. If it is not present, it is inserted, otherwise the device information are updated. At the end of the cycle, the devices that are not active anymore are deleted.

4.6.4 Graph: Logs

The most important and complex set of data was the list of all the logs done by every employee.

Data Retrieval
The most complicated task of this part of the project was to retrieve the list of all logs from Microsoft Azure, since Graph allows to retrieve at most 1000 entry at single HTTP request. To overcome this problem successive requestes was made. The paging system of Azure was managed included the authentication token for the next request in the header of the previous API response, when the next token field was missing it means that it was the last page.
Moreover it was asked to retrieve only the logs done in a specified period, whose values was specified in the `config.ini` file.

```
...
while next_link is not None:
    print("Retrieving information from Azure")
    logs = requests.get(next_link, headers=headers).json()
    insert_logs(conn, logs['value'], file_schema)
    if "@odata.nextLink" in logs:
        next_link = logs['@odata.nextLink']
    else:
        next_link = None
...
```

The above code reports how the paging issue was solved: each time a page is returned, the `nextLink` is saved and used as URL for the next request.

Data Validation
In the same way as explained in the preceding sections, the response of the endpoint was validated, i.e., checked against a `json` file specifically written for the used endpoint.
After the validation, all the date fiels were converted in `date object`, in order to be ready to be used and inserted in the database.

Database insertion
FIrst the table logs was created, with the following structure:

Column Name	#	Data type	Identity	Collation	Not Null
id	1	text		default	[v]
createdatetime	2	timestamp			[v]
user_id	3	text		default	[v]
userdisplayname	4	text		default	[ ]
device_id	5	text		default	[ ]
devicedisplayname	6	text		default	[ ]
city	7	text		default	[ ]
state	8	text		default	[ ]
altitude	9	float8			[ ]
latitude	10	float8			[ ]
longitude	11	float8			[ ]

Figure 5: table logs

Since the logs were retrieved up to a defined number of days and the script may be run everyday, only the new logs had to be inserted in the database. To achieve this result, each entry of the retrieved list was checked against the rows of the db. If the record is not found in the database it is inserted, otherwise it is ignored.

4.7 Docker

After all the code development described in the previous sections, the script was ready and able to retrieve, validate and insert the data in the database. At this point the only problem was that the project was developed on a specific Windows machine, so every command needed to be executed from that machine in order to be able to run properly.
This was a problem since the script needed to be run from any machines, including servers anche Unix environments. To overcome this issue, the whole project was packaged in a Docker container, which create an environment that could be run in every device.

First I had to learn how to use the Docker technology, then I installed Docker Desktop, which is a graphic interface for create and managing docker container. Once I was ready, I created a docker container for my project and I wrote the needed `Dockerfile`.
The `Dockerfile` is the instruction file needed to correctly setup the Docker environment and inside the following parameters was specified:

- **Python**: it was specified which version of Python to use and the standard path of Python in a generic machine.
- **Packages**: the needed packaged for the project were listed in ordere to be installed inside the Docker container.
- **Entrypoint**: using bash instructions, there were specified which commands to execute when the container is started.

Once the container was ready, it was possible to run the wanted script from every device.
In our case, the commads specified in the `Entrypoint` were the ones to retrieve, validate and insert aal the needed data in the database. At the end only one command was run:

```
docker run --rm -it --name python-dev-container -p 1010:1010 python-dev-image
```

as a conseguence of this command the database was populated with the users, devices and logs data.
In the end after the Docker technology was applied to the project, it could be run from any machine and with a single command, all the operations for retrieve the data and populate the database would be executed.

4.8 Data Analysis

Data analysis and the creation of statistics about the available data was the final objective of the project.
There were performed five main analysis:

1. Users who have never done a login with their company account
2. Company devices which have never received a login
3. Company devices which are shared with more than one employee
4. Users who hav done login only in unregistered devices (i.e., devices that are not in the database)
5. Consecutive logs done by the same user in a short amount of time in two different and far locations

Each of the analysis described have a corrispondent command in the console menu. Those commands can be execute with an additional optional parameter which specifies the time duration in which the analysis is performed. If the parameter is not inserted the default interval is ninety days from the current date.

4.8.1 Analysis creation

All the statistics listed before were realized twice: first they were created using only standard SQL without any extention, i.e., each statistic was a single query whose complexity varied from one to the other.
SQL is an achronim for `Structured Query Language`, which is a standardised language that is used to interrogate relational databases.
In a second moment everything done in SQL was realized again using Python, i.e., integrating Python coding with more simple queries to the database.
In this section there will not be any code, but all the SQL and Python files can be find in the project repository.

4.8.2 Unlogged users

The first analysis create was the one dedicated to find all the employees who have never done a login with their company account.
In order to select only the users who have never logged in, it was needed to select the list of users present in the table `users` and not present in the table `logs`.
Initially it was though to use the `NOT EXISTS` or the `EXCEPT` operator, that as the names suggest select the entry available in a table and not in the other one. However, it was found out that these two methods are not very efficient, so it was finally choses to adopt the `LEFT JOIN` operator, which is a kind of join operator which selects the rows of the first table that are not present in the second one. By adopting the join operator, the efficienty of the query incredibly increased.
Since the query was a very simple one, the Python implementation was simply the code needed to run the query.
At the end, after running the corrispondent command of this analysis, the `id` and `displayName` of the selected users were shown as result.

4.8.3 Unused devices

In a second moment it was necessary to create a second analysis, similar to the first one, but regarding the devices. It was indeed needed to find all the devices that have never received a login.
The procedure was similar to the one adopted in the first analysis, so we needed to select only the devices available in the table `devices` and not in the table `logs`.
Since the queory was similar to the previous one, it was directly done using the `LEFT JOIN` to select only the rows present in the first table and not in the second one.
Once again the Python implementation was only the code needed to run the query.
At the end, after running the corrispondent command, the `id` and `displayName` of the selected devices were shown as result.

4.8.4 Shared devices

The third analysis is the one that was created to find all the devices that are shared with more than one employee and all the login done on those devices.
This was a more complex query, since more data manipulations were needed before the expected result coulb be selected.
Itt was needed to select from the logs table only the logs done on devices which had received login from more than one different users.
Firs it was created a subquery in order to select from the table `logs` the `id` of the devices on which have been done access from more than one user.
Then the result of the subquery was joined with the table `logs` in order to select only the logs of the devices whose id was selected before. Finally the result was grouped using the `group by` statement, in order to have not repeated logs done from the same user on the same device. As last step the list of the selected logs was ordered, through the `order by` operator to have a more readable result.
This analysis was more complext with respect to the previous one, but once the logic of the query was leared, everything could be easily written in SQL language. For this reason the Python implementation is, once again, the simple code needed to run a SQL query.
At the end, the result of the main menu command shows the `id` and `displayName` of the shared devices and the `id` and `displayName` of the users who have done login on them.

4.8.5 Logs on unknown devices

This statistics was created to find all the users who have done login only on devices not registered in the company system.
First it was needed a subquery to find all the entries from the table `logs` with the column `deviceId` equal to `NULL`, i.e., all the logs done on an unknown device.
Then the table `users` was joined with the table `logs` in order to have information about both users and logs. From the result of this join operation, there were selected only the rows present in the result of the subquery previously written. By doing this we have obtained only the entries corresponding to the logs done on unknown devices.
Finally, it was required to reuturn information about the users who have done those logs, so from the final result, only the columns `userID` and `userDisplayname` were selected.
The result of the command corrispondent to this analysis are the `id` and `displayName` of the users who have done login on unknown devices.
Once again the Python implementation is the simple code needed to run the corresponding SQL query.

4.8.6 Impossible logs

The last analysis was realized to find all the logs there were physycally impossible, i.e., the logs done in a short time interval and in two different and far location. For example two successive logs done from the same user in a time interval of two minutes and in two location 100 km apart. This example is clearly an impossible log, since it is not possible to travel 100 km in only two minutes.
In the `logs` table, the position of a specific log is given by the latitude and longitude of the device on which the log was done.
PostgreSQL offers an additional module called PostGIS, which is a module that allows to perform spatial queries on the database. This extention is complex and it is used when very precises measurements are needed.
Since it was decided to use only standard SQL and a precise measurement was not needed for the purpose of the analysis, it was chosen to directly calculate the distances between the two points, using the Haversine formula:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cdot\cos(\varphi_2)\cdot\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

where φ_1, φ_2 are the latitude and λ_1, λ_2 the longitude of the two points and r is the radius of the earth.

Since the query needed for this analysis was quite complex, all the calculations were divided among Python and SQL. First it was created a materialized view, which is a view that is stored in the database.
Its purpose is to store all the data of the table `logs`, adding four new columns:

- `prec_time`, which is the time of the previous log done from the same user
- `time_diff`, which is the time difference between the time of the previous and the curret log
- `prec_lat`, which is the latitude of the previous log done from the same user
- `prec_long`, which is the longitude of the previous log done from the same user
- `dist_diff`, which is the distance difference between the location of the previous and the current log

The creation of this materialized view was by far the most difficult part of the anlysis since the complex formula previously described was needed to be written in SQL language. Moreover, in order to calculate the time difference and the distance difference, it was required to access two entries of the table `logs`, so that the selected logs were consecutives and done by the same user. This was not an easy task since in standard SQL is not possible to execute cycles in a query. To overcome this difficulty, I made use of the `lag` Window function. Window functions allow to partition and order the table with respect to a specific column of the table. In our case the logs were partitioned by `user_id` ordered by `createDatetime` in order to access in a consecutive way only the logs done by the same user.
Once the materialized view was created, I performed the remaining calculation using Python. First the time period and the distance used to select the impossible logs were retrieved by the `config.ini` file. The a simple SQL query was written in order to select from the materialized view only the entries whose `dist_diff` and `time_diff` are respectively greater and smaller than the values specified in the `config.ini` file.
At this point all the impossible logs, i.e. the logs done in a time interval shorter than the one required and with a distance between the two points greater than the one specified, had been found.
Finally all the columns of the materialized view were selected in order to give to the user all the information about those impossible logs and about the users who have perfomed them.

4.9 Report Creation

5 Conclusion and Considerations