

Internship Report

Mattia Evangelisti
s268637

June 2022

An internship report presented for the degree of
Computer Engineering

Contents

1	Summary	3
2	Company	4
2.1	Mission	4
2.2	Market Strategy	4
2.3	Economic Growth	5
2.4	Working environment	5
2.5	IT department	5
3	Project Introduction	7
3.1	Used technologies	7
3.2	Standards	8
3.3	Training	8
4	Project	9
4.1	Console menu	9
4.2	Configuration	9
4.3	Database Connection	10
4.4	Migrations	11
4.5	Azure Application	12
4.5.1	Azure Environment	12
4.5.2	Application Registration	12
4.6	Microsoft Graph	12
4.6.1	Authorization and Connection	12
4.6.2	Graph: Users	13
4.6.3	Graph: Devices	15
4.6.4	Graph: Logs	17
4.7	Docker	18
4.8	Data Analysis	19
4.9	Report Creation	19
5	Conclusion and Considerations	19

1 Summary

At the beginning of the academic year I chose to substitute my 3rd 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 necessities 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 collocates 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 through 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 brings in someone else. Through this mechanism, the bill will be proportionally reduced, possibly until zero. On the other hand, by choosing the second option, 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
psycpg2==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.

Once 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 was 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 describes 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.

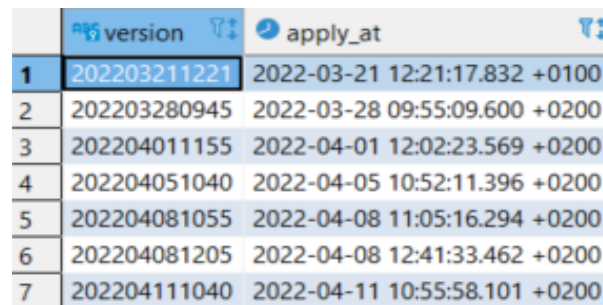
Finally, the cursor is closed and the `conn` object is returned, in order to be used wherever is needed a database connection.

4.4 Migrations

During the development of the project the database was constantly modified: some tables were 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 an intense use of the [migration doctrine](#), applied to PHP. In order to align the 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 learning 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 instructions 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 were 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.

```
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
<code>user_id</code>	1	text		default	[v]
<code>displayname</code>	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` fields 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 only the active devices, but also the devices that are not active anymore, such as old notebooks or phones. 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`, fundamental 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

Flrst 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 consequence of this command the database was populated with the users, devices and logs data.

4.8 Data Analysis

4.9 Report Creation

5 Conclusion and Considerations