UP Discord bot

Henrique Sousa, Pedro Pereira

I. CONTEXT

User Experience is one of the major factors to take into consideration when developing a web app. University's Web apps may not be excellent when it comes to usability. In particular, the web platform *Sigarra* as used by the *Faculty of Engineering of University of Porto* is known to be badly designed and hard to navigate. An alternative has surfaced in the past years called *Uni APP* developed by the *Núcleo de Informática da Associação de Estudantes da Faculdade de Engenharia da Universidade do Porto*
Additionally, as university students, there are a myriad of tasks for which having Quality of Life tools can prove useful. A phone calendar is a simple yet powerful application that many could not live without.
Among college students, there are several communication apps that are used, *Messenger* and *Whataspp*, to name a few. One other that has been gathering a following, particularly between tech-savvy students as tends to be the case for students enrolled in the *Master in Informatics and Computing Engineering* , is *Discord*. Besides standard messaging and video calls, it offers simple and reliable screen sharing, servers, which are used as a meeting point for multiple people, and mastering all its functionalities can provide its users with a better workflow for whatever their task may be. As if that wasn't enough, it is possible to create automated tools, commonly called bots, that expand Discord's base utility.

PROBLEM DESCRIPTION

In today's academic environment, students have the need to juggle between multiple platforms for different tasks: talking to classmates, scheduling project meetings, checking schedules and booking study rooms, for example. However, managing all these activities across different platforms can be incredibly overwhelming and time-consuming.
That's where the need for a more centralized solution arises. Most students are looking for and would benefit of a single platform that can cater to their diverse needs without the hassle of navigating through multiple ways. This way, Discord stands out as an ideal choice not only because of its popularity among students but also due to its exceptional support and ease for developing bot applications.

REQUIREMENTS

First and foremost, the product developed must be a discord bot, like those present in [2].

The following functionalities were identified as requirements and implemented.

- Add people as friends
- Add classes schedule
- View schedule
- Add events
- View events
- Be notified about upcoming events
- Schedule a meeting
- Schedule an office at FEUP's library

SOLUTION

A. Bot

The Discord Bot is implemented through a client made available through the *discord.py* library [3]. It needs a token that is provided by Discord in order to connect. One can either add the bot to their server or interact with it through the Direct Messages. A discord bot will then idle while waiting for an event, which there are many types of. For the purpose of this project, the bot just listens to messages. That is, whenever a message is sent, the bot will wake up and run its *on_message* procedure which is defined by us.
A common way to interact with other bots, and the approach adopted by this project, is to prepend a given prefix when one wants to "speak" with the bot. This allows for a very simple triage of all the messages so that only the relevant ones are completely processed.
If a message starts with the prefix *!*, the bot will then verify if what comes after is valid and take an action based on that. More specifically, it will verify if the message is either a command or if it is the answer to an interaction that has been started with the bot.
There is a fixed list of commands, which can be consulted with the command *!help*. When a message with content *!<command>* is sent, the bot will go through every command and see if there is a match. If so, it will execute as normal. If there is no such match, the user could be trying to continue a previous interaction. However, it might also be that the user used the wrong command. In order to distinguish the two scenarios, we use a stateful approach. This reduces the amount of input that needs to be provided by the user making the bot easier and quicker to use. The state is stored in volatile memory and holds the previous interaction, that is, what the user was doing previously and any accessory data that might be needed for it. using this approach, when a message addressed to the box is received, if it is none of the commands, the previous interaction will be resumed. The content of the message is checked and still validated within the context of the interaction. The user is allowed to cancel the previous interaction with *!cancel*. Any valid command will also cancel the interaction (and start a new one if that is the case).

B. *Storing Uporto schedules*

To make life easier for users of the application, the user should have access to University of Porto's schedules and be able to choose amongst them instead of manually inputting the classes they have and their information. We leveraged an existing project that scrapes the web for the schedules [7]. We followed the running instructions and managed to fetch the information. This data follows a relational schema and is stored in a *.db* file. As such, we used the *sqlite3* module [6] in order to interact with it and fetch the data during the execution. First, we developed a simple API to interact with a general SQL database. On top of that, we build an application-specific API that enables us to fetch the needed information during the program's execution. No changes to the database are made when the bot is running. The application-specific API operates mostly by using the underlying API to acquire the results of *JOIN* operations. After that, it knows which class of object is are to be returned, so it parses the multiple columns into multiple objects and returns the wanted object to the application. This object belongs to a well-defined class, and, therefore, eases the manipulation of data for storing in the users' database or for being displayed.

C. *User data*

In order to store the user data we used mongoDB [4] in a container, to which the application connects at the start. It is a document database. We chose this database paradigm as user data can be easily conceptualized as a document with various fields. For most of the operations, this allows us to access the corresponding document from the user id, and then fetching, adding, updating or removing a field as needed.

D. *Functionalities*

D.1) *Username*

Users area able to add their UP username by using the command:

- !add_username <upXXXXXXXXX@up.pt>

D.2) *Friends*

We developed a friend system, that allows users to connect to each other by allowing them to see their friends' schedules. This are the available commands:

- !add_friend <@user>
- !friend_requests
- !accept <int>
- !friends_list
- !remove_friend <@user>

To start a friendship, a user must use the !add_friend command, tagging the person to be friended using the built-in tagging discord system, separated by an empty space. The other user should then check their list of friend request by using the command !friend_requests, where the bot will respond in discord with a list of the user's incoming friend requests which are associated with an integer as seen on the following example:

1. user2
2. user3

Then, in order to accept an incoming friend request a user should use the command !accept followed by the integer associated with the person to be friended. In order to check its own friend list, a user should use the command !friends_list, which will make the bot answer with the list of the user's friends. Finally, if a user wishes to remove a friend, he can do so by using the command !remove_friend, followed by the tag of the user to be unfriended.

D.3) *Schedule*

- !add_schedule
- !view_schedule

There are two ways to add classes to the schedule. The first is by selecting classes from the set of classes in *University of Porto* I-B. The second is by manually adding a class. By choosing the command *!add_schedule* the bot will present a set of choices.

To add a class manually the user needs just select the appropriate choice and follow the instructions. If the user provides valid class details, day of the week, time, and duration, the class is created and stored alongside the user data. Selecting a class is a little more complicated, as the user needs to navigate the menus to choose their faculty, course and course_unit. The use of a document database, instead of a standard relational database, prevents the need for multiple tables by adding the data hierarchically, as can be seen below.

```
faculties: [
    {name: icbas,
    full_name: Instituto de Ci ncias
        Biom dicas Abel Salazar (ICBAS),
    courses: []},
    {name: faup,
    full_name: Faculdade de Arquitetura,
    courses: [
        {id: 1,
        acronym: MIARQ,
        name: Mestrado Integrado em
            Arquitetura,
        course_units: [
            {name: Sistemas Construtivos
                Tradicionais,
            acronym: 50154C5,
            id: 258,
            year: 4,
            enroll_year: 2023,
            semester: 1,
            classes: [
                {name: 2LEIC03,
                lesson_type: T,
                id: 258,
                location: B001,
                day: 1,
                start_time: 480,
                        duration: 60,
                        professor: 3045099
                        }
                    ]
                    }
                ]
                }
            ]
            }
]
```

Listing 1: Schedule data structure

The data to add new entities (faculties, courses, course units, or class) to the user data uses the University of Porto's schedules which were previously obtained I-B.

To view the set of selected and manually added classes, the user can simply use the command *!view_schedule*.

D.4) Add events

Event commands are the following:

- !add_event <name> <DD/MM/YYYY> <HH:HH>
- !events
- !events delete <int>

Events serve as personalized reminders for users, specifically designed to help them remember important commitments, tasks, or occasions at specific dates and times. At the set time of an event, a message is sent to the user, notifying the event's start. In order to create an event, a user should use the command !add_event followed by the name of the event and the time at which the event will occur. Regarding time specification, various time formats are supported and the only mandatory argument is the day of the event, while the hour can me omitted or not.

D.5) Schedule Meeting

The commands involved are

- !schedule_meeting
- !deschedule_meeting
- !view_meetings

A user can add a meeting to their data. A meeting must necessarily be with someone else (otherwise an event would suffice), which must be provided with the command. A meeting must specify a date, time, and duration. Trying to schedule a date will provide feedback on the availability of the participants, including oneself. In order to check the availability all time based information is aggregated in a single data field, with a type classfier

The user can see the meetings they have and remove one of them with the command *!deschedule_meeting*

The user can view their meetings with the command *!view_meetings*.

D.6) Reserve Office

The commands involved are

- !reserve_office
- !view_office_reservations
- !cancel_office
- !add_number

The bot allows the user to reserve an office at *FEUP*'s library. First, the user must have already input their name with the command *!add_number*. When calling *!reserve_office*, the bot will ask for a date, time and duration, and optional motivations and observations might be provided. This replicates *Sigarra*'s procedure. A confirmation will be required before attempting to reserve the office. To do so, the bot will create an authenticated session in *Sigarra* using the developer's credentials which must have been given in the *.env* file. Afterwards, it will do a post request to the same page a user would to try and reserve a room [5]. If no error occurs and the room is available, an id which is used to do the confirmation of the request with another post request. Feedback is sent to the user on whether or not the reservation is successful. If it was, the aforementioned id is stored as user data. When the user tries to cancel the reservation with *!cancel_office* the aforementioned id is used for the post request that will cancel the reservation. *!view_office_reservations* allows the user to view the current reservation (only one reservation is allowed).

Testing

For testing, we developed acceptance tests for the most important features of our project, namely:

- Emulating sending message commands from other users for debugging purposes
- Creating, deleting and viewing events while also simulating user input mistakes
- Creating/scheduling, removing and viewing meetings
- Adding name
- Adding number
- Reserving, cancelling and viewing FEUP's library offices
- Adding class information (faculty, course, course unit and class) in order to generate weekly schedule

Challenges

D.7) User schedule

Some user workflow was envisioned at the start and had to be changed as the work was developed. For example, and since we took the UNI app as inspiration, we thought of allowing the user to have their schedule available without any menial work, just by giving their authentication details. As we are focused on security, we didn't feel comfortable incentivizing users to send their authentication information directly through Discord. Another solution that we looked into was asking users to give their authentication cookies, limiting the effect of a possible security mishap. However, this solution didn't prove reliable as the session was bound to terminate which would require the user to do the non-trivial work of fetching the necessary data again. Instead, we settled on acquiring all of Porto University's classes and respective schedules and allowing the user to choose amongst those.

D.8) *Notifications*

This challenge arose when we tried to have the bot send a notification to a user when an event they had added was coming due. Adding the closest event to a data structure and fetching the data once it was time for the respective notification was easily achievable. However, knowing when to and effectively launching the notification gave me some trouble. We thought of having a timer thread, a thread that will only execute when the timer has come to its conclusion, sending the message. However, since sending a message is an asynchronous function, the whole thread would have to be asynchronous. We found that starting such a thread was impossible unless asyncio [1] was used to start a new event loop. Again, we faced problems as that would mean that the bot's main thread and our *notification thread* would have different event loops. This meant, in practice, that variables created in an event loop could not be used in another. Namely, the *client* variable that holds the bot couldn't be invoked to send the message.
After some research, we discovered that the Discord library had a proper way to run an asynchronous function alongside the main bot loop, which is the solution we used.

D.9) *Duplicated data*

The data obtained from the *uporto-schedule-scrapper* wasn't completely correct. We noticed that the number of courses of a given degree was three times the actual number of courses, with two sets of duplicates. Additionally, some class schedules were duplicated once. There might not have been usability problems as the data seemed to all be there. The only disadvantage is presenting too much data and repeated data to the user.
In order to fix that issue, we had to manipulate the relational database. First, we took some time to identify the problem properly. As it turns out, the database keeps a *last_updated* column that was the only one that was different between the duplicated rows (besides the primary key). Somewhere during the data-gathering process, maybe due to human error, the same data was fetched multiple times, only with different timestamps. In order to solve this, we had to write a query for the relevant tables that would only delete the rows for which there was an equivalent row already. Here's an example of a query that was made

```sql
select from schedule as a where exists (
    select * from schedule as b where a.
    day=b.day and a.start_time = b.
    start_time and a.location = b.location
     and a.lesson_type = b.lesson_type and
     a.is_composed = b.is_composed and a.
    professor_sigarra_id = b.
    professor_sigarra_id and a.
    course_unit_id = b.course_unit_id and
    a.class_name = b.class_name and (a.
    composed_class_name = b.
    composed_class_name or (a.
    composed_class_name is NULL and b.
    composed_class_name is null)) and a.id
     > b.id);
```

Listing 2: SQL query

Notice how a.id > b.id is used to delete the duplicated records with the highest ids. Also, for this specific table, since *composed_class_name* can be *NULL* a more complex condition is used to guarantee that both null records are deleted.

Conclusion

The developed project, in the form of a discord bot, fulfills in providing an alternative way to organize college life and interaction with some parts of *Sigarra*. It presents a simple, yet intuitive, interface that will be familiar to avid users of Discord. By navigating through it the user can manage their schedule, events, meetings, and even library offices. From a more technical scope, it uses both the relational and document database paradigms in order to take advantage of existing software and remain simple to use. It also offers an opportunity for more functionalities to be added in the future, including by different parties, since the requirements for running a bot are very low. In conclusion, we hope to have implemented a fruitful application that solves some of the problems outlined in Context by using a platform that we enjoy and that is widely spread.

References

[1] asyncio —asynchronous i/o —python 3.12.1 documentation. https://docs.python.org/3/library/asyncio.html. Accessed on January 6th, 2024.

[2] Discord bots. https://discord.bots.gg, . Accessed on January 6th, 2024.

[3] discord.py · PyPI. https://pypi.org/project/discord.py/, . Accessed on January 6th, 2024.

[4] Mongodb. https://www.mongodb.com/. Accessed on January 6th, 2024.

[5] Reserve library room. https://sigarra.up.pt/feup/pt/res_recursos_geral.pedidos_valida. Accessed on January 6th, 2024.

[6] sqlite3 —db2.0 interface for sqlite databases —python 3.12.1 documentation. https://docs.python.org/3/library/sqlite3.html. Accessed on January 6th, 2024.

[7] NIAEFEUP/uporto-schedule-scrapper: Python solution to extract the courses schedules from the different faculties of UPorto. used to feed our timetable selection platform for students, TTS. https://github.com/{NIAEFEUP}/uporto-schedule-scrapper. Accessed on January 6th, 2024.