UniTN Booking Bot

Alex Pegoraro
Department of Industrial Engineering
University of Trento
Trento, Italy
alex.pegoraro@studenti.unitn.it

***Abstract*—UniTN Booking Bot is an Internet of Things application built on top of the Telegram Bot API, that allows students to seamlessly book and manage appointments with companies during fairs like the "Industrial Engineering Days".**

This document describes the system architecture and the implementation choices made both on the Telegram front-end and the SQLite back-end.

***Index Terms*—Internet o Things, Telegram Bot, SQLite.**

I. Introduction

Universities often organize events in which students have the opportunity to interact with companies trough conferences as well as individual meetings. This allows the students to secure internships and explore future working opportunities, while the companies can generate awareness and increase the pool of candidates for their specialized positions, often difficult to find in the traditional job market.

Managing this fairs can be very cumbersome for a University, and the user experience is often over-complicated for both the parties involved. UniTN Booking Bot aims to simplify the advertising and booking process of an event, using exclusivity a Telegram Bot. Thanks to it meetings will be arranged with a few taps on an app we use every day rather than having to pass through complex sites with multi-factor authentication, guaranteeing a faster and more efficient communication. This will translate into an higher user satisfaction and a streamlining of the event monitoring process.

While the Telegram interface manages the networking connection and guarantees the association between a user account and a physical person, it does not ensure data persistence. This feature is achieved introducing an SQLite database as a back-end component.

II. The Database

A. Why SQLite

SQLite [4] is a lightweight implementation of a relational database particularly suitable for IoT applications and natively available in Python. I selected a relational database because the data used in this project are completely structured: every record has a precise schema that is unlikely to change over time. Furthermore, the amount of data is not enough to require horizontal scaling, especially if we consider the fact that after the end of every fair all its events will be removed from the database. On the contrary, I expect an high volume on transactions, especially *read-only* and *update* ones, another aspect to prefer a relational database.

B. The Table Structure

Every instance of Unitn Booking Bot will have its own database file, with a set of tables organized as in Fig. 1. The core of the system is the `events` table. Every time a company wants to organize an event, e.g. a conference from De Longhi or a set of student interviews by Open Move, they can issue a command to add a record to this table: its name together with a brief description will be stored in the Bot, and visible by any user that will request an event list.

To improve the organization, the visibility of the events is limited by a fair scope. The `fairs` table contains logical groups of events, like "Industrial Engineering Days" or "ICT Days". In this way, a user can visualize only events related to a selected fair, avoiding to erroneously book appointments with the wrong department. This also shortens the number of records the Bot needs to send when requested an event list, improving the readability of the answer.

In case the event is an open conference, it's enough to publish it with room and time in the description, and wait for the interested to come. However, many events are organized into small time slots dedicated to a single person, that needs a booking, such as in the student-company interviews. The `slots` table accomplish this need. Together with the `event` record, the company will publish a set of `slot` records liked to it, representing the various time slots available for booking. Initially, such slots will be *free*, i.e. associated with a `NULL` user, but then the students will use the Bot to link them to their own account, indicating a booking.

The last table is the `users` table, that unlike the other ones do not use auto-incremental primary keys, but identifies each user trough the unique Telegram chat ID of the chat between them and the bot. This way access control is seamless: a user is allowed to modify an event only if the chat that issue the command has a chat ID equal to the `owner_id` of the event. At the same time booking an event requires just to set the proper `slots` field to the current chat ID, without the need to navigate the `users` table beforehand.

Finally, I would like to highlight the role of the `username` field in this table: it contains the Telegram username, i.e. the unique string starting with @ that allows users to find each other on Telegram. Since a company has access to all the slots linked to their event, they can ask the bot to provide them the usernames of all the students that booked an appointment with them. With this feature they can send direct messages to them,

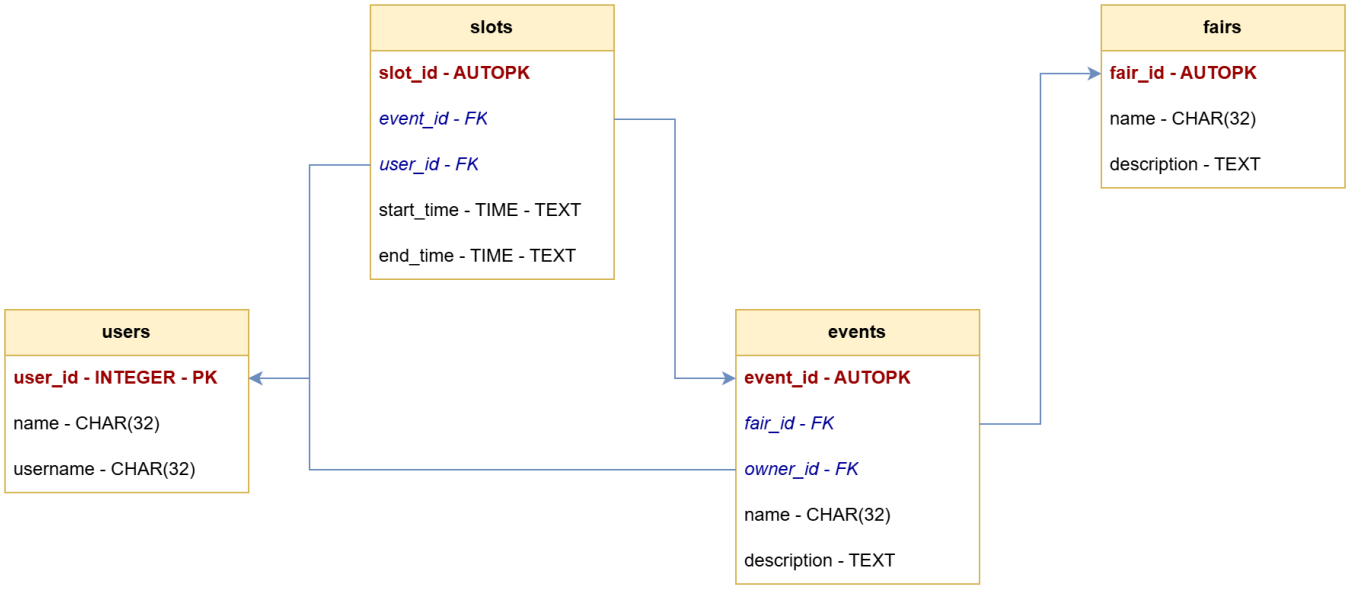

Fig. 1. Schema of UniTN Booking Bot's Database.

for example to give them additional information regarding the meeting or promptly warn them in case of troubles or unexpected event cancellation.

III. The Telegram Bot API

A. The Infrastructure

Since the API has not been explained in the lectures, I will use this section to briefly introduce it. The Telegram Bot API [1] is an HTTP-based interface that allows personal applications to interact with a Telegram server. It is compatible with many programming languages, and in my case I decided to use the implementation for python, called `python-telegram-bot` [2] [3].

Bot accounts are created contacting "The Bot Father" (@BotFather) application from your telegram account. However, such Bots have neither computational power nor persistent storage, it is a duty of the developer to provide them together with the program logic.

Programs like this project are run on a local machine and use the API to repeatedly send HTTP requests to a Telegram server. They authenticate themselves through a *token* string, given to you by the Bot Father during the account creation. The server answers with all the new messages from user that contacted the Bot. Now, you can handle the message with your own Bot logic and use the API to send the answer back to the server, that will route them to the correct chats.

B. Stateless Message Handling

From a programmatic perspective, the Bot logic is achieved through the definition of *Message Handlers*, i.e. pairs of *filters* and python functions. The idea is that each incoming massage is checked by a filter, and in case of a positive match the correspondent function is executed. Such function will be allowed to access both the message content as well as the chat parameters. *Command Handlers* are a particular type of handlers that match messages starting with a slash followed by a desired string, which identifies the command.

Bots interactions are not limited to pure text: you can present the user a keyboard of buttons, each one associated with a *callback* string sent to the your program in case of tapping. Callbacks are not processed by message handlers, but are intercepted by *Callback Query Handlers*; however the concept is exactly the same: if the callback matches a certain *pattern* then the correspondent python function is invoked and will process it.

C. Stateful Message Handling

Sometimes we want a function to be executed only in certain situation, even if the message may match the associated filter in other situations. The typical case is that we want a function to be execute only after another function has been executed beforehand.

The way to achieve this behavior is trough *Conversation Handlers*. They wrap message handlers (and also callback query handlers) assigning them to a certain *state*. When the bot is in a conversation it will check only the handlers associated to the current state, neglecting the others. Furthermore, the functions associated with such handlers will have to return the next state to assign the bot after their execution, or return the end of conversation command.

It is important to notice that initially the Bot will not be in a conversation and will check none of the handlers internal to it. The conversation needs to be triggered by a special set of message handlers, defined the *entry point* of the conversation. If any of these handlers is match, then the bot will go into the state specified by their function.

D. Consideration on Message Handlers

When using a *Callback Query Handler* to send the user a new keyboard, you can set the callbacks of the buttons to be their own predefined string but with the current callback string appended to it. Like this you are forwarding the information to the next *Callback Query Handler*, and you will be able to infer the sequence of button pressed with a simple parsing of the final callback.

On the other hand, when using *Message Handlers* you have access only to the user's last message, because humans will not append their previous message to the current one. So, the only way you can reconstruct the full message sequence is to have a global variable in your bot that associates user IDs with their message sequence, and update that variable every message. It is clear that this approach introduces a notable overhead.

This is the reason why in this project *Conversation Handlers* contain just *Callback Query Handler*, or in case they have a single *Message Handler* it is put at the beginning of the conversation: this way the entire set of user actions can be reconstructed looking just at the final callback, without the need for global variables.

IV. The Doxygen Documentation

This project is designed to be effectively deployed by the University of Trento, and for this reason I supported it with an extensive documentation. I created it with Doxygen, a documentation software based on function comments. Each function I have implemented embodies a special comment string, that describes its parameters, the return values and its aim in general. At the same time, each file begins with a section enunciating the rationale of the entire module itself.

The documentation is obtained compiling all the comments in the files according to a set of parameters defined in a `Doxyfile` in the root folder. The output is a full HTML site, ready to be exposed to the internet. In my case, I used the *GitHub pages* of my repository to host it at the following address: https://alphanightlight.github.io/UnitnBookingBot

V. The Bot Implementation

A. Project Structure

The main file of the project is `bot_main.py`, which instantiate the Bot, defines the handlers and exposes the project to the internet. The python functions associated to the handlers are imported from the three files in the `handlers` folder.

The `.env` file contains the project secrets and environment variables. The Bot token is stored there, to avoid it being hard-coded and increase security and portability. The other important role of this file is to store the database location. Having this information in a global variable is the best solution to ensure different scripts are orchestrated to address the same database instance. The easiest way to create a database with the schema of Fig. 1 is to run the script `create_db.py`. On the other hand, a comprehensive set of functions to interact with a database of this kind is defined in the three modules of the `utils` folder.

Interact with a database through a text message interface like Telegram is the key point to bring this service to a non-expert user base, as well as ensuring access control to the resources and providing ubiquitous availability. However, in case some problems arise a more powerful approach is needed: the file `create_db.py` import functions with direct access to the database. Its operations are granular at record level and have full visibility of the data; this means it can quickly fix almost any data inconsistency, but it may introduce a lot of new ones if used unconsciously.

There is another important usage of this file: `fairs` can't be created or removed by the Bot, since no handlers are defined for this task, so the only way to manage them is the direct access to the table. This policy have been chosen for the following reason: events are managed by companies, while fairs are managed by the university itself. Hence, there is no reason to let companies indiscriminately generate their own fairs, it shall be the owner of the Bot that decides to which group an event can belong.

Another extremely important feature of this project, is the extensive documentation. A full Doxygen documentation [6] of every function has been generated in the `docs` folder, and is also publicly available at the Git Hub page of the Repository. At the same time the `README.md` provides details on the proper project installation, while `info/cheatsheet.md` is a concise summary of the most common commands the user will encounter.

B. Bot Commands

The UniTN Booking Bot handlers can be divided into four categories. On the first one, we have commands like `/fairs` and `/events`, which return information about a required record in the homonymous table. Similarly, `/whoami` returns the user record associated with the chat from which the command was issued. In particular, the Chat identifier is the crucial field to communicate to the bot owner in case some user problems needs to be fixed with direct access. These three commands are all read only with respect to the database.

The next set of commands are meant to be used by the students in order to book their appointments. `/book` allows the user to select a *free* time slot, and mark it with their user identifier. Other users will no more see the slot as bookable, and the company will a username pared with the slot. In case the user wish to cancel a booking, the `/unbook` command will restore the slot's user to `NULL`, making it available again. Finally, it may be in the interest of the student to remind which slots they booked; trough `/mybookings` they can see all their time slots with the associated events.

The company-side commands are centered in the manipulation of events. To insert a new record in the `events` table it's enough to run `/publish`. This data will have an empty description, which can be quickly updated with `/changedes`. Next step is to add new free slots, using the `/newslot` command. I would like to highlight that SQLite store time values as time strings in the ISO-8601 format, so many checks are done on the provided input dates to ensure

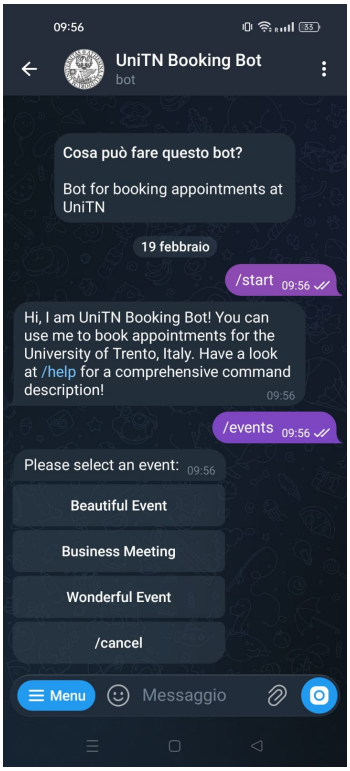

Fig. 2. Screenshot of a possible selection display for the `/events` command.

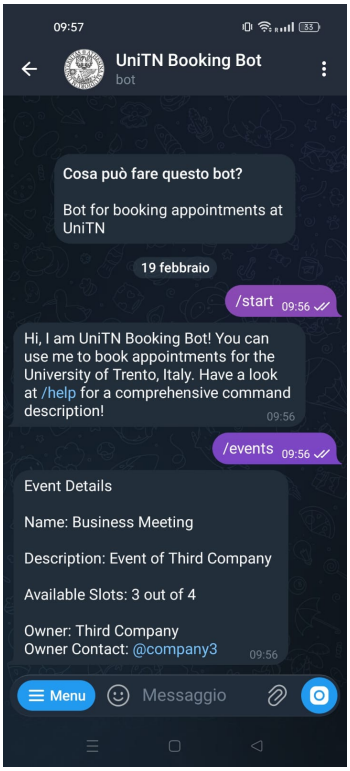

Fig. 3. Screenshot of a possible final output for the `/events` command.

they comply with the standard. There is no limit to the number of slots a company can assign to their events.

It is possible to delete from the database single slots as well as entire events, using `/deleteslot` and `/deleteevent` respectively. In the second case all the slots associated to the event will be deleted as well. Note that after a slot has been deleted it is no more possible to know the user that booked it, so in case you need to warn the user of a sudden event cancellation please do so *before* the actual deletion of the slot.

Also the companies have the possibility to monitor the bookings of their slots. With `/myevents` the company can visualize all the slots associated to an event, know if they are free or booked, and also access the Telegram username of the person making the reservation.

The last set of handlers are the ones dealing with the default commands `/start` and `/help`, together with the handler for invalid commands and the one for free text. All of them simply respond with an hard-coded message, that is list of command descriptions for `/help` and a brief presentation of the Bot for the others.

VI. User Journey

This section will explore the typical life cycle of a fair, both from the perspective of the students and the perspective of the companies.

A. Event Publication

The prerequisite to be able to publish an event, is that the university has already instantiated a fair trough the script `edit_db.py`. If this is the case, the company will issue the `/publish` command and add the event to the required fair. Immediately after that, they will update its description with `/changedes`. If the event is open this step is finished, but if the event requires a reservation then they will publish an amount of slots equal to the desired participants, using `/newslot`.

B. Event Booking

The students will monitor UniTN Booking Bot trough the `/events` command. It will list all the events for a given fair like in Fig. 2, and tapping on them will produce additional information, such as the number of available slots and the username of the person that published the event. Fig. 3 displays the output of such command for a sample database.

When they have decided the events they are interested in, they can select the slot to book issuing the `/book` command. If after a while they no more wish to attend, they are expected to run the `/unbook` command to release their slot. Changing the slot is not directly supported, but is achievable with a simple unbook and rebook.

C. Event Monitoring

Now, it is time for both the sides to check their statuses. Students can do so with the `/mybookings` command, whose output is presented in Fig. 4. They will have an overview of all the slots they booked, together with their timings and the event associated with them.

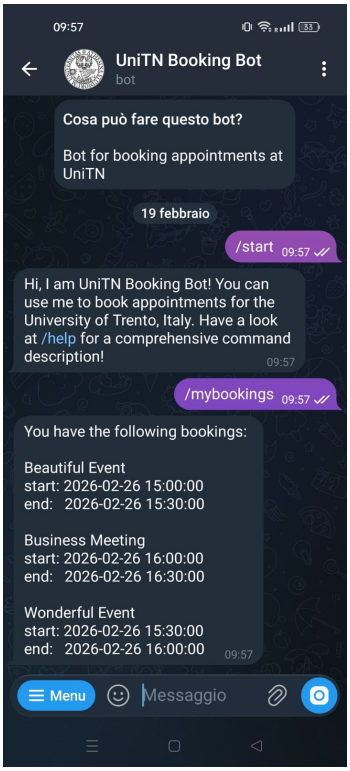

Fig. 4. Screenshot of a possible output for the `/mybookings` command.

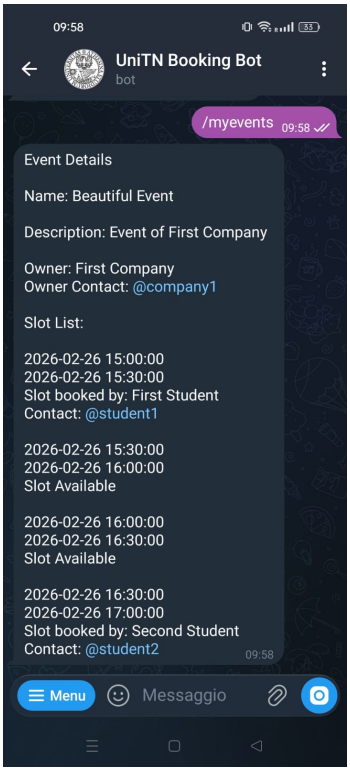

Fig. 5. Screenshot of a possible final output for the `/myevents` command.

On the other hand, `/myevents` will provide the companies with a detailed report on a selected event of their own, such as the one showed in Fig. 5. They will know all the slot associated with it, if it is booked or not and in the former case who is the user that reserved it.

D. The Meeting

At the moment of the meeting, the company can ask the student for authentication. The process is very simple: the user issues `/whoami` on his Bot, and the username provided as an answer is compared to the username associated to the current time slot as the company can see with `/myevents`. In case there's a match, the chat shown by the student is exactly the one that booked the slot and so they are in the right place.

After the event is over, the company can use `/deleteevent` to remove the event as well as all the associated slots from the database. This will clean it from outdated information and will avoid it to grow unbounded, justifying the selection of SQLite. in case the company forgets to do so, the Bot administrator can manually do it with `edit_db.py`.

VII. Future Works

A possible extension to this project would be to implement the following feature: the Bot may automatically send a notice half an hour before the event, both to the student and the company. The naive idea would be to use the built-in timer callback of Telegram Bot, but this solution suffers to main problems: on the one hand it is not persistent, in case of even a Bot failure the scheduled message would be lost even if the problem is solved before the message due time. On the other hand, if the bot do not fail than the message will be inevitable, even if the user changes booking or the event is canceled.

The solution I have in mind to bypass these issues, is to store the notices as records in a new table of the database. The Bot would check this table periodically, and schedule messages for the next period. For example, it may check the table every half an hour, and when it wake up at 16:00 it would schedule a timer callback for all the events from 16:30 to 17:00. Then it will wake up at 16:30 and schedule all the messages from 17:00 to 17:30, and so on. This could be an nice idea for the next year's project.

Other Resources

This report provided an accurate description of the system's architecture and the design choices of the project. Other resources you can read include:

- The repository `README.md`, concerning the installation and run of the project on a local machine.
- The full project documentation, rooted in `docs/index.html` and accessible at the following GitHub page: https://alphanightlight.github.io/UnitnBookingBot/
- The command cheatsheet, located in `info/cheatsheet.md`.

Furthermore, the References section of this report includes links to the official documentations of all the APIs

used by this project. Such links are also replicated in the file `info/api_reference.txt` for who will not read this report.

ACKNOWLEDGMENTS

This project has been realized for the course "Laboratory of Internet of Things", held by prof. Davide Brunelli, prof. David Macii and prof. Matteo Nardello at the *University of Trento*.

REFERENCES

[1] Telegram Bot API, last access 18 Feb 2026: https://core.telegram.org/bots/api
[2] GitHub repository of python-telegram-bot, last access 18 Feb 2026: https://github.com/python-telegram-bot/python-telegram-bot/wiki
[3] Official documentation of python-telegram-bot, last access 18 Feb 2026: https://docs.python-telegram-bot.org/en/stable/index.html
[4] Official documentation of SQLite, last access 18 Feb 2026: https://sqlite.org/index.html
[5] Quick reference for SQLite, last access 18 Feb 2026: https://www.geeksforgeeks.org/python/python-sqlite
[6] Official documentation of Doxygen, last access 18 Feb 2026: https://www.doxygen.nl/manual/index.html