TAW PROJECT REPORT

made by Alberto Barison (901146@stud.unive.it)

project colleagues (895898@stud.unive.it , 903407@stud.unive.it)

The service architecture

Our web service is built upon a Model-View-Controller architecture, where each of the three components (front-end, back-end and database) is containerized separately, even though the whole application can be runned with one command.

As requested, our front-end is built on the Angular framework, while the back-end supports a set of APIs which are REST-style compliant, running on the express framework. All the data is stored and handled on a relational database (Postgres) along with proper functions and constraints policies.
The web service offers many functionalities, spanning from authentication of different kinds of users to booking of lights that are designed by different airline companies, along with token expiration mechanics and so on…
Here follows, in detail, the various parts of our application.

THE DATABASE

Overview

For this project we agreed on the use of a relational database, therefore picking Postgres as the best choice. We thought that the agility of managing documents in noSql databases such as Mongo would have meant more difficulties on another side, that is maintaining a coherent consistency of data that are related to each other, in particular, how different actions on the client side would effect at row level in the database as time went on.

Along with Pg, we have picked an ORM, Prisma in our case, to make the developer experience seamless but also less error-prone, thanks to some key features Prisma provides, which did fit well in our backend environment. Even though Prisma is part of the backend, it's been developed at its core by who developed the database and then lent its maintenance to the member responsible for the backend - then passed to one another when some arrangements where needed.

Interaction between the parts

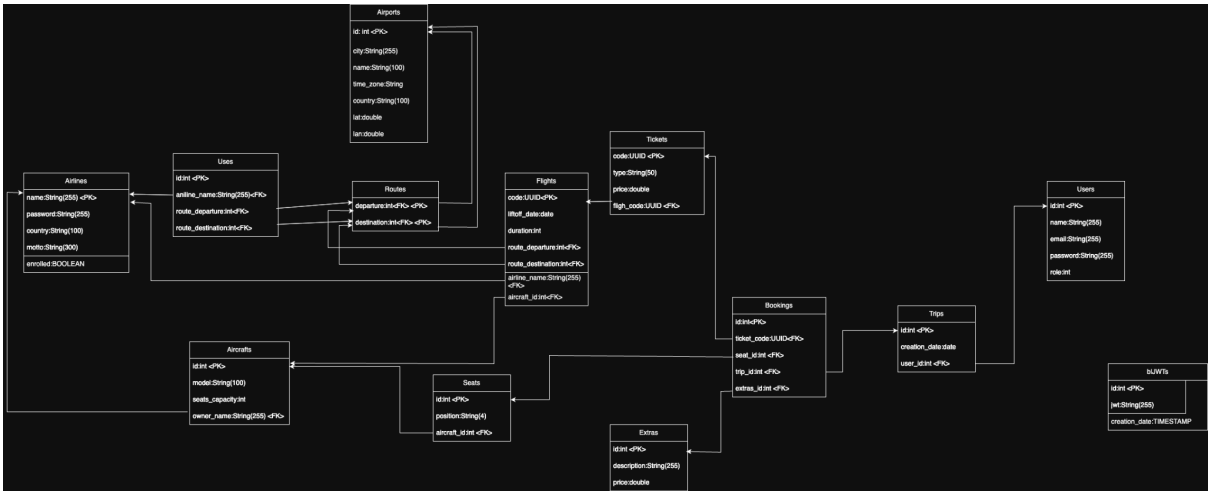
The database would have been like the bottom of a deep well of data: If we want to assure some water to the top, then ensuring a clean transition of it is a must-have, both on the developer and the user experience (especially).
This is where Prisma comes in, standing as a sort of middleware layer between backend and database, with the ability to *introspect* the database schema but also query it with no raw SQL at all.
This doesn’t mean no SQL was ever written. It means that, once the database was well defined (gonna talk about it later on), no future feature adding, operation testing and whatnot would have been slowed down by the SQL syntax. This is how we deliver water to our users without the need to reinvent the bucket every time.

Through proper shell commands, prisma client - crucial for interacting with the database - is generated and ready to talk with a Pg running container and to deliver the data to and from the database with little or no overhead.
In order to work, Prisma needs to “point” to the right database specs (port, db name,...) which are specified in the .env file in the backend folder.
It should be clear from the start that our pg container loads some data to begin with at the first spin.

Our database philosophy and the data model

A strong importance, while defining the database schema, was given to the need to set a division of entities for building fine grained behaviours, while the relations would have been built on the base of common logic when imagining the user experience, as a sort of consequence.

Here follows the ER schema, on which the SQL dump was built



As you can see, different relationships here are one-to-many and just a few many-to-many.
For identification purposes, IDs were mostly preferred and UUIDs come across in rare occasions. The philosophy behind this quite intricate division of entities is really handy when you immagine use cases that need certain triggers to operate for a specific objective - keep in mind - without compromising the existence of records and so on; changes in our tables are properly handled.

For instance, when a row in the *Booking* table no longer points to a seat *and* a ticket? Well, the user may need that booking just to show that a payment was made; on the other hand, a Booking record without a reference to a Trip (therefore a user) will still be useful to an Airline for analytics purposes.

However, when those (instances, rows of) tables mentioned above, that are referenced via a foreign key, are not linked anymore *all together*, a trigger ad hoc will remove the booking record, since it's pointless to keep it in the database.

Certain policies were also adopted to not waste available space: Notice how the *Seat* table does not use any "taken" attribute, but rather serve available seats to the end user by hiding those who are not.

A brief description of our table follows suit:
Users hold basic information along with a particular attribute, the role, which allows us to distinguish between the admin and the others.

Airports are interviewed only by the admin, while the *Routes* serve as a bridge that connects these airports. Crucial to know, a route can be used by many airlines and no route duplicate can be created: If you need it and it exists already, use it.

Aircrafts are exclusively handled by airlines, and their *Seats* are sort of dependent on those aircrafts from their generation (automized) and their deletion (also automized).

Flights and *Tickets* are the last two entities that belong to the airline field of actions. When creating a flight, an airline will need a route on top of which will be established, as well as a ticket with a certain cost that will be offered to the user. Flights and Tickets are also strongly related: none of the two can exist without the other.
Bookings servers (relationally speaking) as a connection point between the tables where users and airlines operate. Bookings of course are interviewed by the user (not admin) and each of it is enclosed with another one at most in a trip, since two bookings can be offered to reach a certain destination.

lbJWTs is a blacklist that holds JWT that once belonged to now deleted users. Every record in this table will be deleted the day after its insertion. The purpose of this is ensuring the invalidation of the (now invalid) token.

THE BACK-END (with APIs description)

Overview

As we can appreciate from the folder structure, the backend holds, based on its functionalities, sections that serve the database interaction and definition (*database* and *prisma* folders), middlewares and utils (reusable) modules and, most importantly, the APIs.
Starting from user authentication, which is a mechanism which is largely required across our application, we opted for a join use of JWT (Json-Web-Token) and local storage (browser side) to implement routes, therefore resources protections.
In order to protect the user from malicious attacks that exploit theft of the token, we choose to set an expiration time. The token itself contains a special integer that distinguishes between normal user, admin and airline. Certain endpoints rely on the token not just for authorization purposes, but for information such as the airline name.

Before looking at certain endpoints that best express our REST-style APIs, we stop by to appreciate how prisma made our life easier, thanks to its key features such as *prisma introspection* and the dry but clean syntax which, besides the database definition phase and certain fundamental changes, allowed us to move with ease in the definition of proper query-functions: from an idea to its realization with little effort!

Our APIs

The APIs we defined follow the REST style, focusing on simplicity, describing resources rather than the actions made on them (which are expressed by the kind of the HTTP request). The main roots of our endpoints are

- Accounts
- Airlines
- Navigation
- Bookings

Since they are the main actors, interacting with each other.
The basic endpoint comes in the form of

/api/{resource}/{sub-endpoint}

Here's a list of the endpoints designed:

Accounts

POST {{serverEndpoint}}{{serverPort}}/api/users/user

GET {{serverEndpoint}}{{serverPort}}/api/users/user

POST {{serverEndpoint}}{{serverPort}}/api/users/login

DELETE {{serverEndpoint}}{{serverPort}}/api/users/user

GET {{serverEndpoint}}{{serverPort}}/api/users/accounts

DELETE {{serverEndpoint}}{{serverPort}}/api/users/accounts/:id

Airlines

POST {{serverEndpoint}}{{serverPort}}/api/airlines/invite

POST {{serverEndpoint}}{{serverPort}}/api/airlines/enroll/:invitationCode/:airlineName

POST {{serverEndpoint}}{{serverPort}}/api/airlines/login

GET {{serverEndpoint}}{{serverPort}}/api/airlines/airlines

POST {{serverEndpoint}}{{serverPort}}/api/airlines/aircrafts

GET {{serverEndpoint}}{{serverPort}}/api/airlines/aircrafts/:airlineName

DELETE {{serverEndpoint}}{{serverPort}}/api/airlines/aircrafts/:aircraftId

POST {{serverEndpoint}}{{serverPort}}/api/airlines/routes

DELETE {{serverEndpoint}}{{serverPort}}/api/airlines/routes/:routeId

GET {{serverEndpoint}}{{serverPort}}/api/airlines/routes

GET {{serverEndpoint}}{{serverPort}}/api/airlines/flights

GET {{serverEndpoint}}{{serverPort}}/api/airlines/flights/pending

POST{{serverEndpoint}}{{serverPort}}/api/airlines/tickets

Navigation

GET {{serverEndpoint}}{{serverPort}}/api/navigate/airports

GET {{serverEndpoint}}{{serverPort}}/api/navigate/airports/:departure/routes

GET {{serverEndpoint}}{{serverPort}}/api/navigate/airports/:destination/incoming-routes

GET {{serverEndpoint}}{{serverPort}}/api/navigate/routes/path?from=1&to=16

POST {{serverEndpoint}}{{serverPort}}/api/navigate/routes

POST {{serverEndpoint}}{{serverPort}}/api/navigate/flights

GET {{serverEndpointDev}}{{serverPort}}/api/navigate/flights

Bookings

GET {{serverEndpoint}}{{serverPort}}/api/bookings/seats/:flightUUID

GET {{serverEndpoint}}{{serverPort}}/api/bookings/tickets/:flightUUID

POST {{serverEndpointDev}}{{serverPort}}/api/bookings/trips

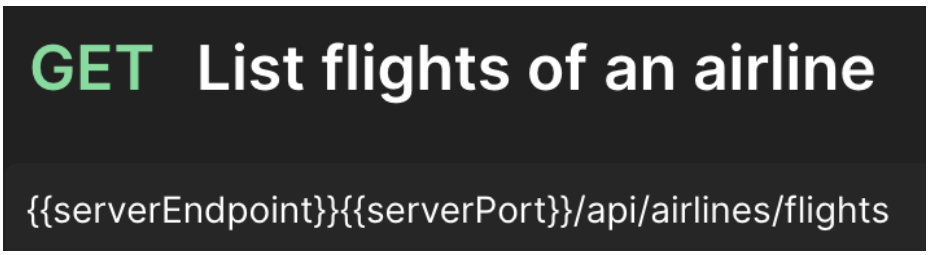
GET {{serverEndpoint}}{{serverPort}}/api/bookings/trips/:tripId

Other endpoints

GET {{serverEndpointDev}}{{serverPort}}/api/utcTime

Other features

Let’s look at an example which can summarize at best what we built:

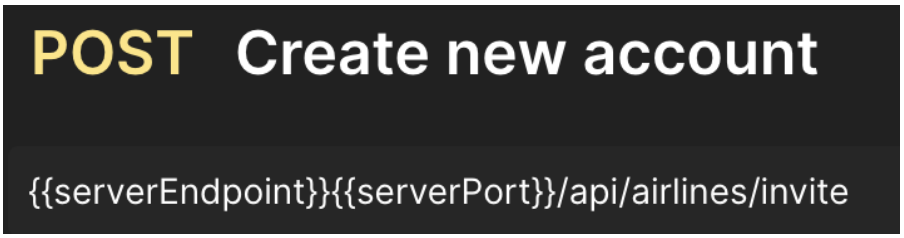


The purpose of this endpoint is pretty self explanatory, but we should point out that the Bearer token authorization mechanism is not the only thing the JWT, passed in the request header, can serve. In fact, the same JWT provides the airline's name, crucial for fetching the right resources, along with the obvious protection of the route.

Here's the JSON returned when the request succeeded


```
{
    "message": "List of flights retrieved successfully",
    "flights": [
        {
            "code": "187f190c-505e-425a-bf49-bebcbd9eaca4",
            "liftoff_date": "2025-10-14T22:00:00.000Z",
            "duration": 91,
            "route_departure": 4,
            "route_destination": 1,
            "aircraft_id": 5,
            "lifoffTimeZone": "Europe/Rome",
            "liftoff_dateLOCAL": "2025-10-15T00:00:00",
            "aircraft_details": {
                "model": "Boeing 777",
                "seats_capacity": 396
            }
        },
        {
            "code": "6ef89afc-ca82-4ed7-b76c-101b5fb10c41",
            "liftoff_date": "2025-10-04T22:00:00.000Z",
            "duration": 91,
            "route_departure": 4,
            "route_destination": 1,
            "aircraft_id": 5,
            "lifoffTimeZone": "Europe/Rome",
            "liftoff_dateLOCAL": "2025-10-05T00:00:00",
            "aircraft_details": {
                "model": "Boeing 777",
                "seats_capacity": 396
            }
        },
```

Most importantly, this endpoint perfectly adapts to the core concept behind REST-style. We are providing no information about the kind of action we are performing, while writing out loud what information is interviewed, i.e. a set of flights linked in some way or another to an airline.
It can be said something similar about the following endpoint



…which can make us appreciate how well integrated these APIs are in our server, going hand to hand with middleware functions. Just a brief look to the different functions called for this endpoint

```
const authMiddleware = require('../../middleware/auth');
const checkAirlineEnrolled = require('../../middleware/checkAirlineEnrollment');
const validateJsonRequest = require('../../middleware/validateJsonRequest');
var airlineRouter = express.Router();

airlineRouter.post  ('/invite', validateJsonRequest, authMiddleware, controller.createAirlineNewAccount);
```

Finally, as for the user authorization and authentication of any user in the platform, besides the JWT hashing, signing etc… We implemented functions that validate the data and sanitize it, in order to ensure proper protection of the server.At last, our server implemented several microservices in order to handle corner cases, like serving error displays, redirecting when a session has expired and so on…

THE FRONT-END

Overview

Our front-end application was built on the Angular Framework, which is very popular in the world of SPAs.

During the whole process of development we appreciated how its large pool of modules and the detailed documentation served to quickly give us a set of tools to easily implement the desired page structure and underlying logic.
Thanks to modularity, a *divide et impera* policy made the structuring of the page and its components closer, in a logical sense, to the nature of the data stored in the database, therefore easier to present to the front-end user.

A strong separation was made between the components (small and reusable pieces) and pages (still components, responsible to host more components, like the canvas of a painting); with this approach, we found ourselves many times dealing with fetching and subsequently passing the data, returned by the server, to specific components, each of them with a chunk of information. This is where inputs, outputs, components routing and modularity offered by Angular came handy.

We structured the whole frontend in order to give the easiest user experience possible, taking inspiration from modern flights-booking web applications.
The main pages span from the homepage with its flight search bar, to the section designed to customize your ticket and buy it, and finally to the sign/login and personal areas, different from one kind of user to another.

We also placed, just like in the backend, some global variables that are widely used across the platform, such as IPs and ports.

Our components

Here follows the list of components, pages and routing:

Routing (app.routes.ts)

- /
  → HomepageComponent
- /signin
  → SigninComponent
- /forbidden
  → ForbiddenComponent

- /admin-dashboard (protetto, ruolo admin)
  - /airline-registration → AirlineRegistrationComponent
  - /airlines-management → AirlinesManagementComponent
  - /airports-management → AirportsManagementComponent
  - /users-management → UsersManagementComponent
  - / → redirect /airline-registration
- /homepage
  → HomepageComponent
- /airline-enrollment
  → redirect /forbidden
- /airline-enrollment/:invitationCode/:airlineName
  → AirlineEnrollmentComponent
- /airline-login
  → AirlineLoginComponent
- /user-registration
  → UserRegistrationComponent
- /homepage-airline (protetto, ruolo airline)
  - /routes-management → RoutesManagementComponent
  - /aircrafts-management → AircraftsManagementComponent
  - /flights-management → FlightsManagementComponent
  - / → redirect /routes-management
- /server-error
  → ServerErrorComponent
- /flights-display
  → FlightsDisplayComponent
- /profile-customer
  → ProfileCustomerComponent
- /ticket-booking
  → TicketBookingComponent

Pagine principali (pages/)

- admin-dashboard/
  - admin-dashboard.component.*
  - airline-registration/airline-registration.component.*
  - airlines-management/airlines-management.component.*

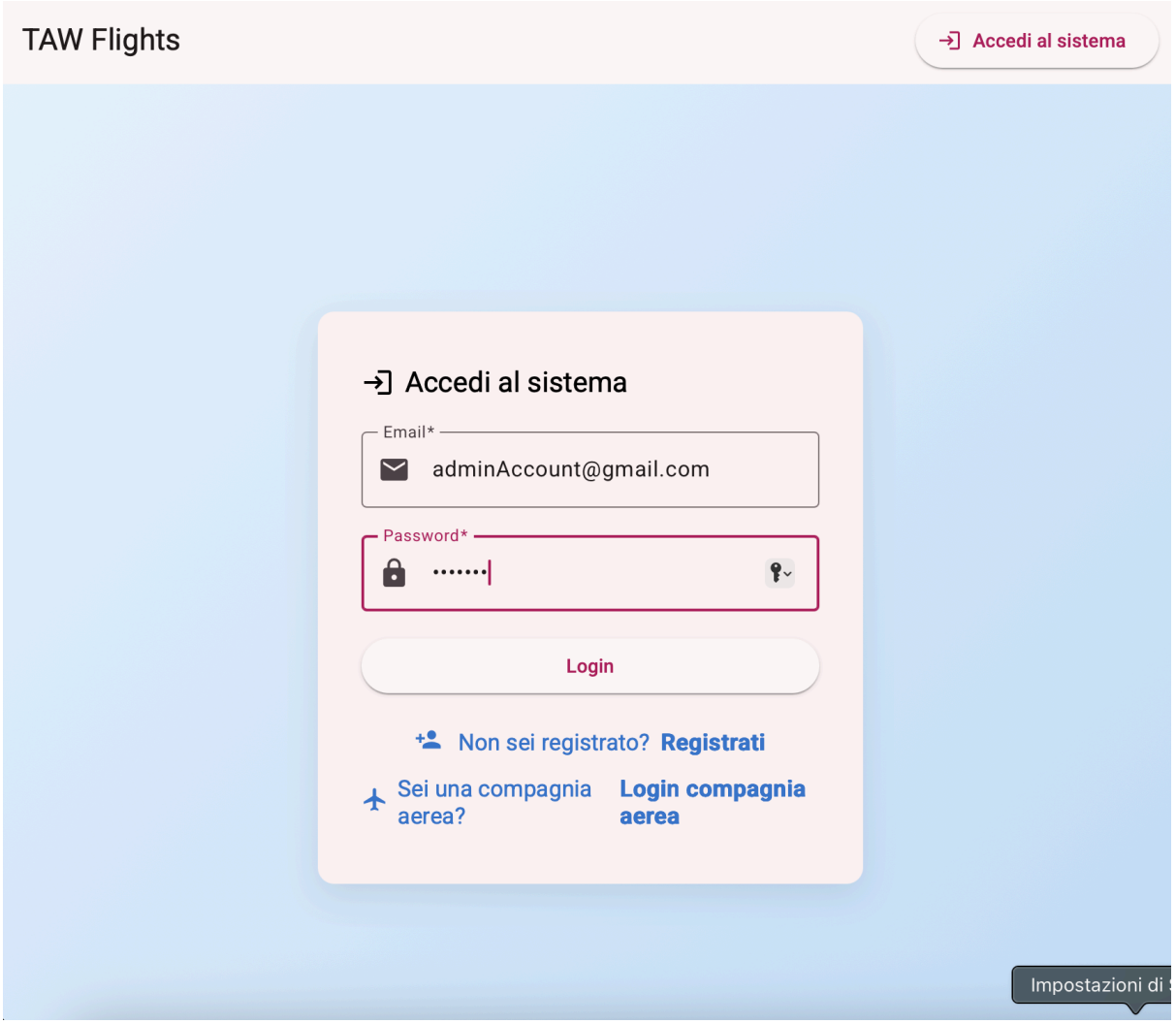
    - airports-management/airports-management.component.*
    - users-management/users-management.component.*
- homepage/
    - homepage.component.*
- homepage-airline/
    - homepage-airline.component.*
    - routes-management/routes-management.component.*
    - aircrafts-management/aircrafts-management.component.*
    - flights-management/flights-management.component.*
- profiles/
    - profile-customer/profile-customer.component.*
- flights-display/
    - flights-display.component.*
- ticket-booking/
    - ticket-booking.component.*
- forbidden/
    - forbidden.component.*
- signin/
    - signin.component.*

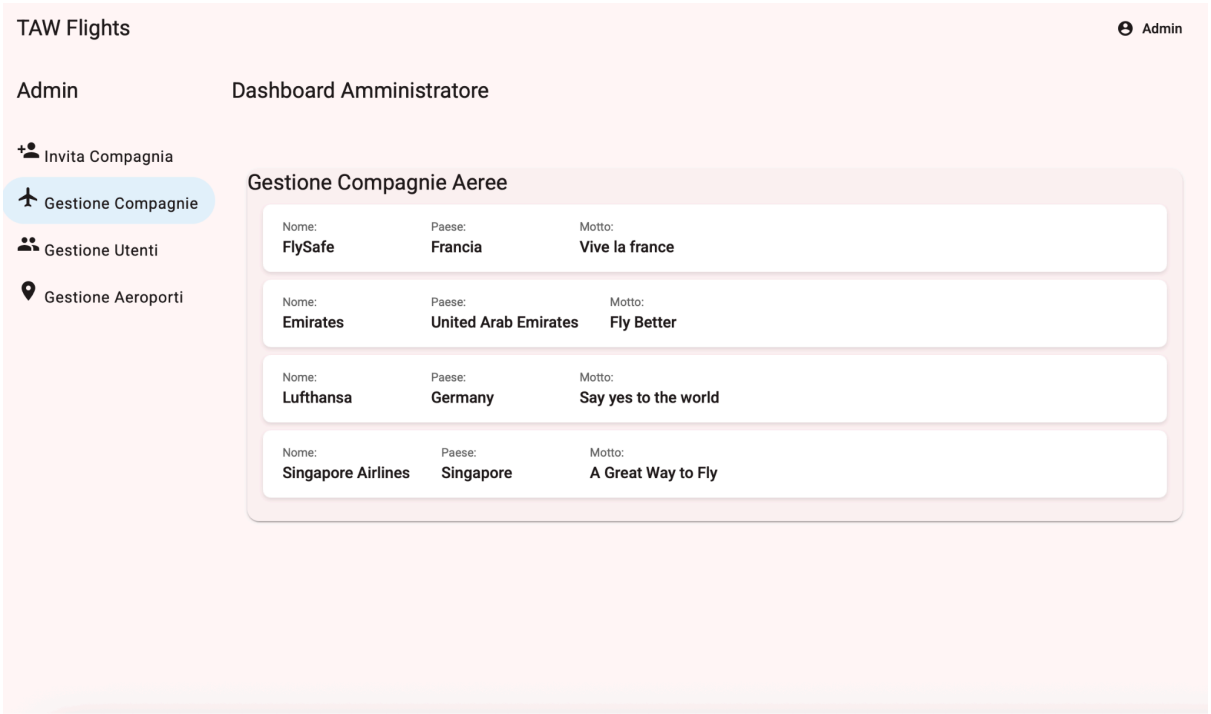
Componenti principali (components/)

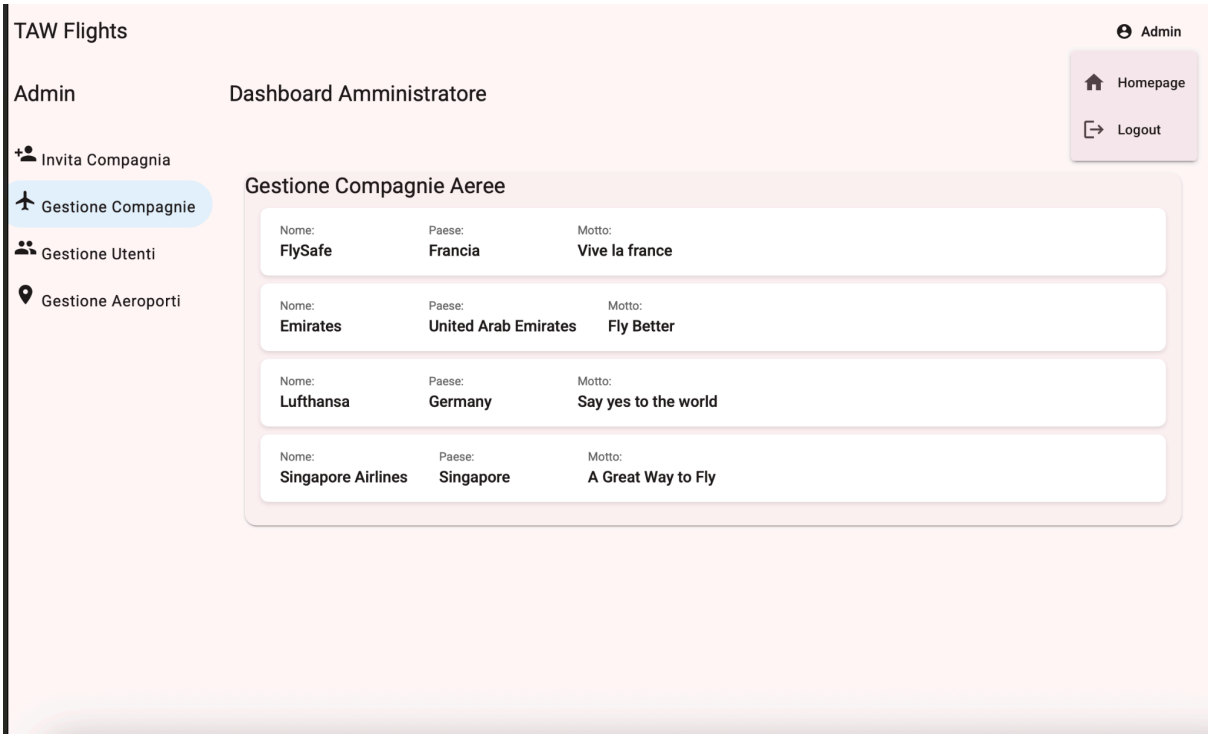
- toolbar/toolbar.component.*
- flight-search/flight-search.component.*
- flight-list/flight-list.component.*
- flight-card/flight-card.component.*
- airline-flight-card/airline-flight-card.component.*
- route-card/route-card.component.*
- aircraft-card/aircraft-card.component.*
- airport-card/airport-card.component.*
- trip-card/trip-card.component.*
- map/map.component.*
- map-routing/map-routing.component.*
- user-card/user-card.component.*
- airline-card/airline-card.component.*

Some examples from our application

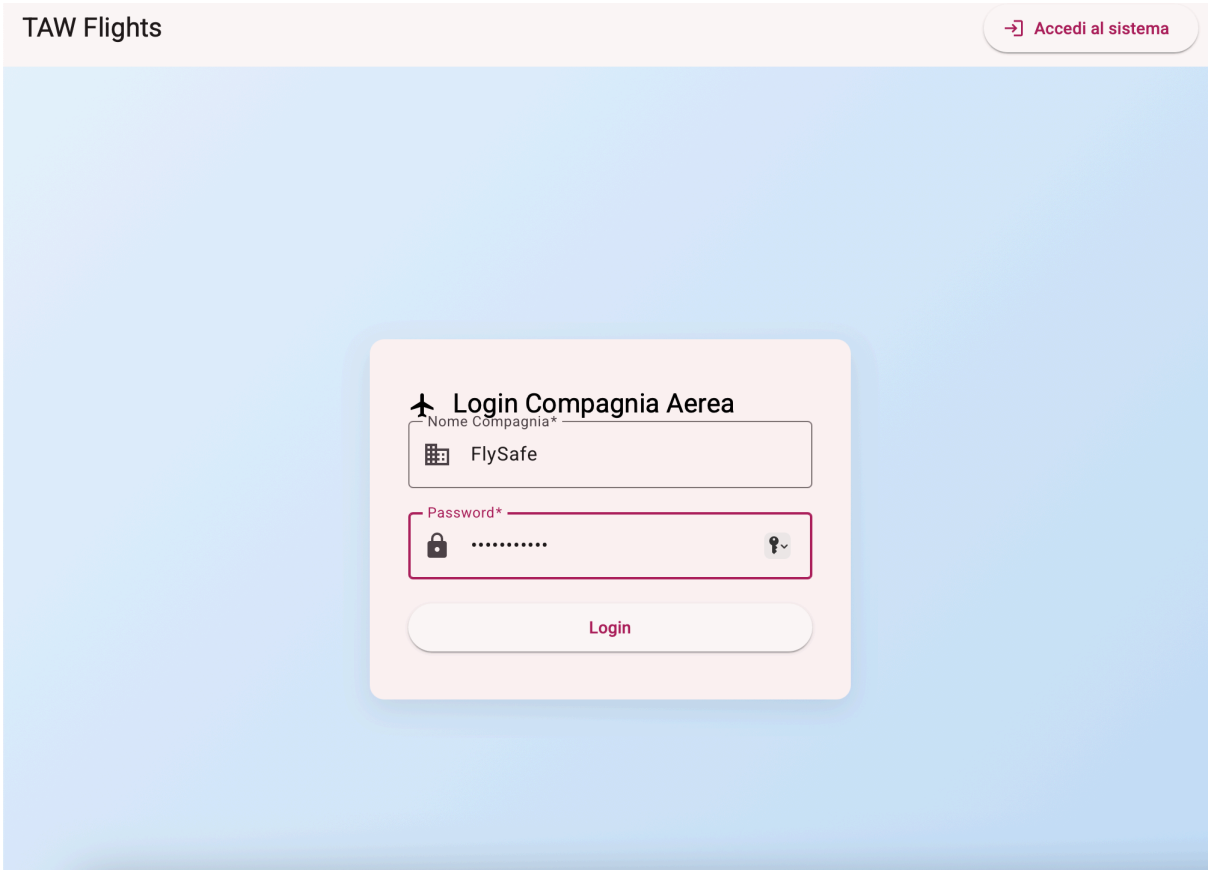
possible travelling to and from the admin dashboard

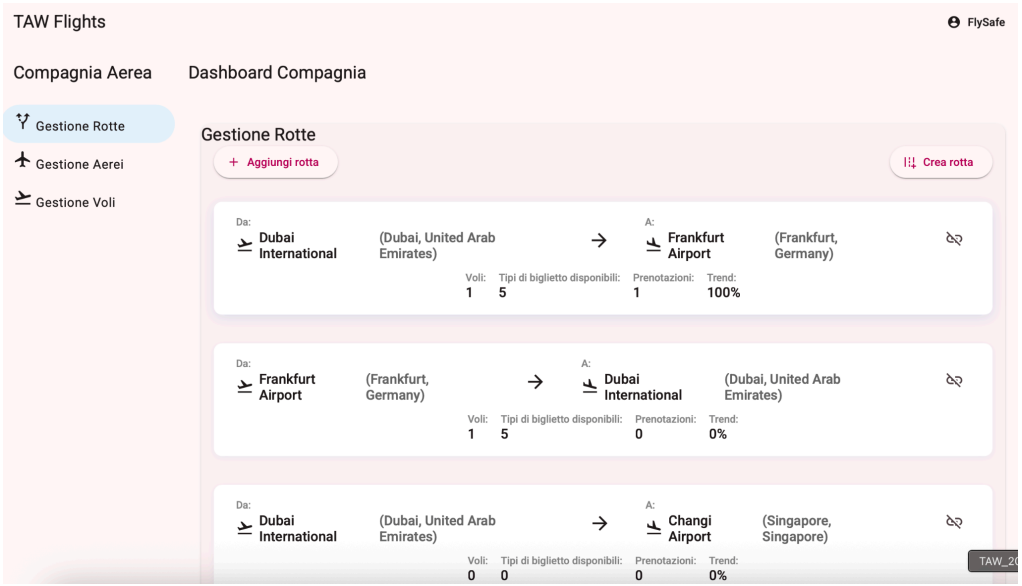


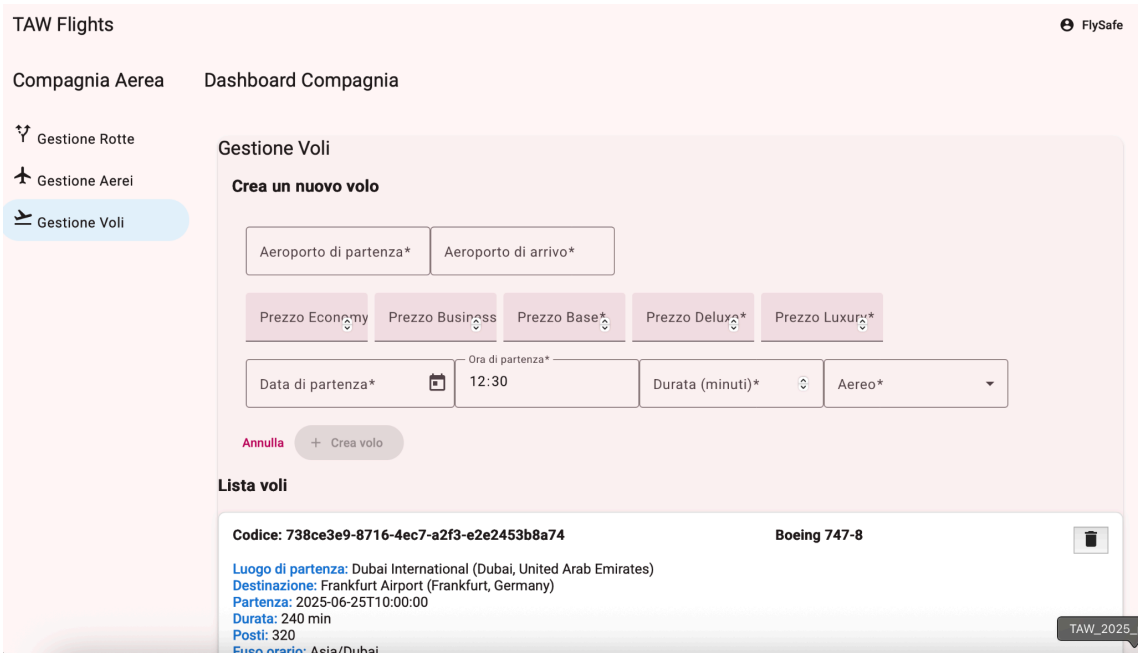

TAW Flights
Admin
Admin
Dashboard Amministratore
Invita Compagnia
Gestione Compagnie
Gestione Utenti
Gestione Aeroporti
Gestione Compagnie Aeree
Nome: FlySafe
Paese: Francia
Motto: Vive la france
Nome: Emirates
Paese: United Arab Emirates
Motto: Fly Better
Nome: Lufthansa
Paese: Germany
Motto: Say yes to the world
Nome: Singapore Airlines
Paese: Singapore
Motto: A Great Way to Fly


TAW Flights
Admin
Admin
Homepage
Logout
Dashboard Amministratore
Invita Compagnia
Gestione Compagnie
Gestione Utenti
Gestione Aeroporti
Gestione Compagnie Aeree
Nome: FlySafe
Paese: Francia
Motto: Vive la france
Nome: Emirates
Paese: United Arab Emirates
Motto: Fly Better
Nome: Lufthansa
Paese: Germany
Motto: Say yes to the world
Nome: Singapore Airlines
Paese: Singapore
Motto: A Great Way to Fly

Possible travelling to and within the airline homepage

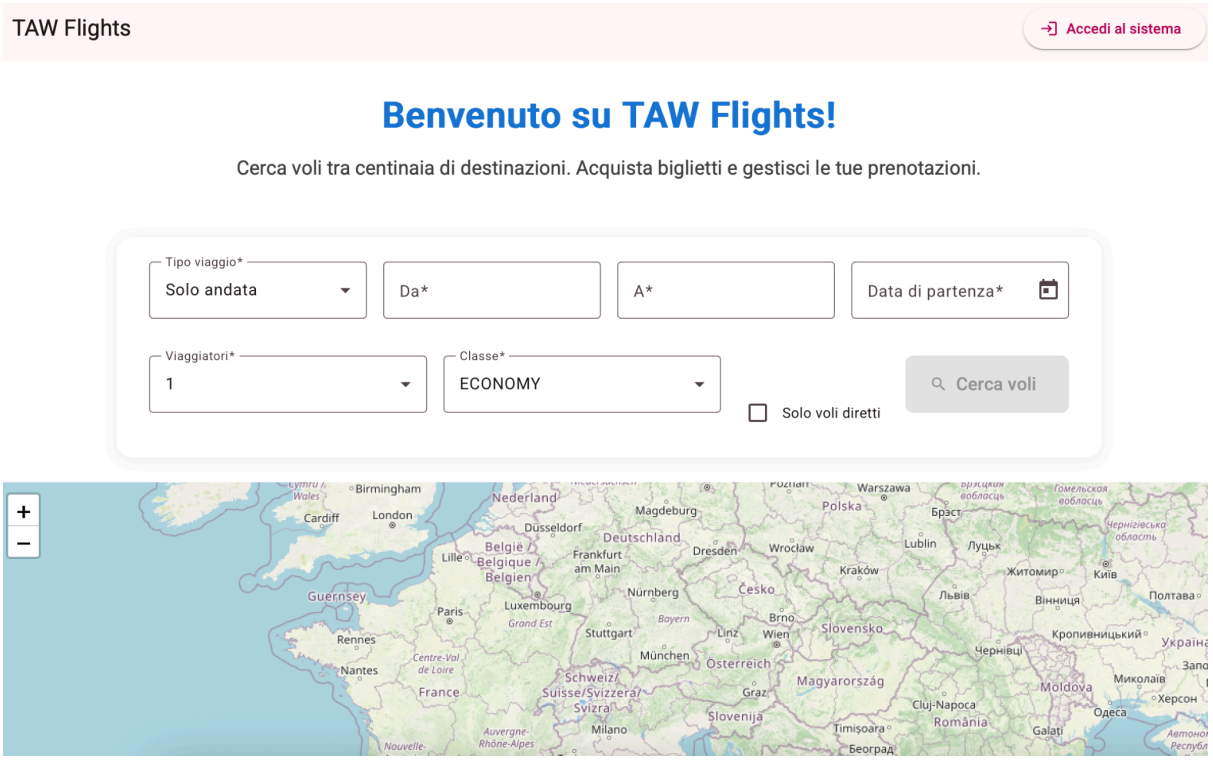


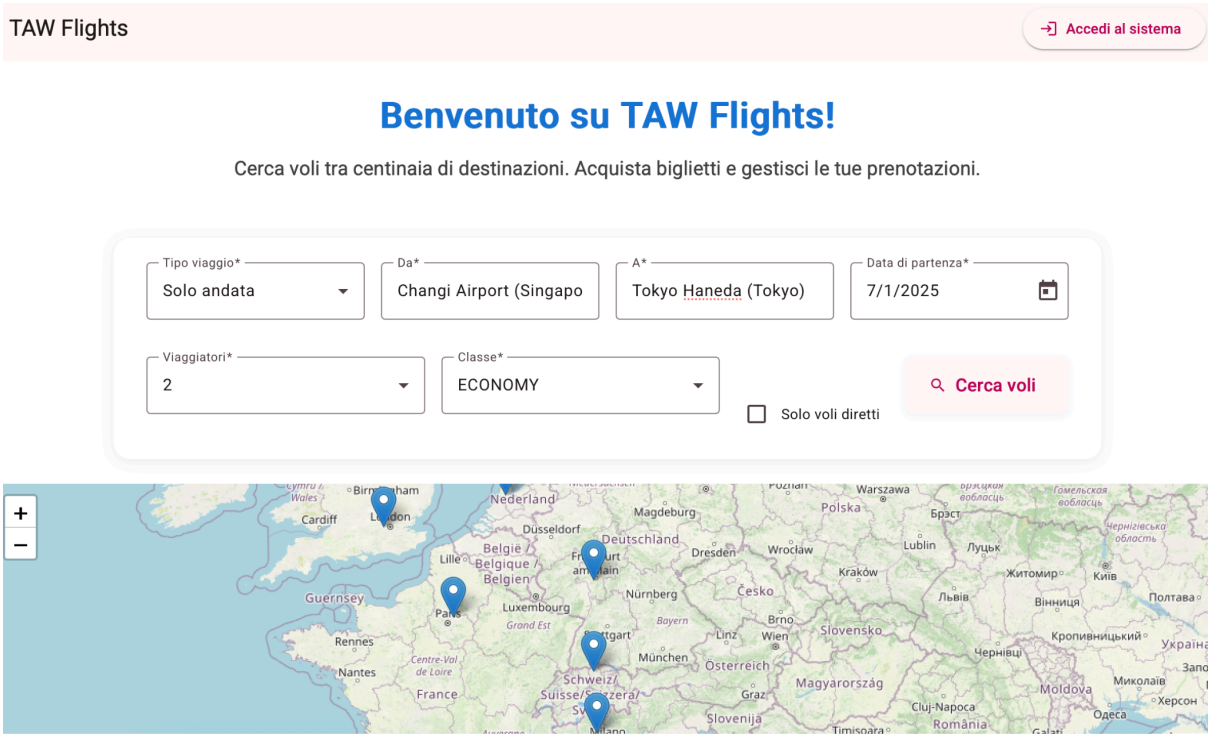


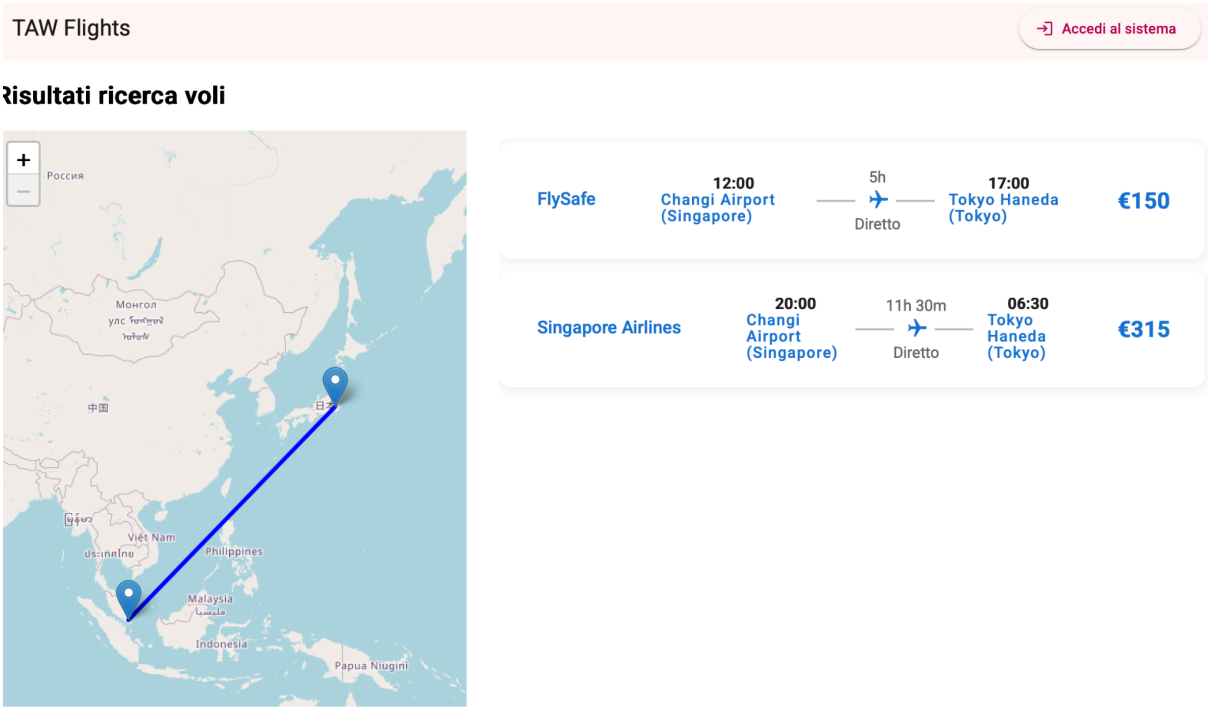


(As we can see from these last images, we presented some statistics that can be useful to the airline).

Possible ways of finding a flight







The tools we used

The whole development process of this application was supported by a strict but essential set of tools, like Github for synchronizing the workflow of the whole codebase (along with GitHub desktop), postman for designing and testing of APIs, Docker for containerizing the app components, PgAdmin4 for building and testing the database maintenance and Visual Studio Code, for writing the code itself.