DS 5110 Final Project

Ethan Morgan, Rohit Barve

Abstract

For our final project, we designed and implemented a flight reservation system. We consider three types of users: administrators, customers, and pilots. For tooling, our database is implemented in MySQL, the data was generated using Pandas DataFrames, and the application front end uses a Python Flask application. The system allows users to sign up, view their flights, and book new flights. Pilots can sign up, view the flights they are piloting, and sign up for new jobs. Administrators can view the entire system, including a set of statistics on most popular airports and most active customers.

Introduction

Over the course of the project, we iterated several times on the schema and relations. After much discussion, we landed on our final design. We started out with a complex design where a customer had an identifying relationship with another table called users. Eventually, we found that it made more sense to only store information on a user once they signed up for our system. Similarly, we started out with far more relationships between tables. For example, we had relations called seat_in_flight and sits_in that connected customer, seat, and flight. These tables were ultimately removed, opting to include their primary keys in either the trip or flight relation.

We decided that there were logically three categories of users of our system. Customers should be able to see all flights, their own user details, and their own trips. Pilots should be able to view all flights, their own jobs, and their user details. Finally, administrators should be able to view customer IDs but not other customer details, all flights, all seats, all trips, and all pilot and attendant attributes.

Database Design

The database entity-relationship diagram is shown below in **Fig. 1**. The main relations for the database are customer, pilot, attendant, airport, seat, and flight.

The relation **customer** contains the primary key **customer_id**, as well as some identifying information including name, address, and date of birth. A customer can book a trip on a particular seat of a flight, assuming the flight is not already full.

The relation **pilot** contains the primary key **pilot_num**, and also identifying information about the pilot. A pilot can sign up to pilot a particular flight, assuming that the flight does not already have a pilot signed up, which is reflected in the relation **pilots_flight**. This relation is similar to the relation **attendant**, with the difference that **attendant_num** serves as primary key for this relation. Similar to pilots, attendants can sign up to attend a particular flight, reflected in the relation **attends_flight**.

The relation **airport** has primary key **airport_code**, which is a 3-letter identifying acronym for an airport. Additionally, locational information for each airport is kept, including address, city, and state. The relation **gate** is a weak entity under airport, having additional attributes terminal and gate_id.

The central relation in this database is **flight**. It has primary key **flight_num**, and connects the relations gate, seat, trip, customer, attendant, and pilot. This relation is connected to gate twice, as a flight departs from a gate at one airport and arrives at a gate at another airport. A constraint imposed here is that the airport_code for arrival must be different from the airport_code for the departure airport; a flight cannot fly from an airport to the same airport. The flight relation also connects to the relations pilot and attendant through the relations pilots_flight and attends_flight, respectively. The only difference here is that a flight can only have one pilot, but can have multiple flight attendants. Finally, the flight relation is connected to customer through trip and seat. A customer can have a trip booked, which has a seat on the flight. One constraint here is that a customer can only book a seat on flight if the seat is not already booked by somebody else.

Though we initially planned on gathering the data to populate the database through a flight API, we were not able to find a suitable API that fit our needs. We instead generated synthetic data using Pandas DataFrames in Python. The airport table is the only table that represents real data; we took the top 10 airports in the United States and found their airport codes and other attributes through Google. From there, we randomly generated the rest of the tables.

For all instances of IDs, we calculated a random numeric or alphanumeric string of the appropriate length. For customer, pilot, and attendant names, we defined a pool of male first names, female first names, and last names. For each person, we randomly chose a gender, and picked out an appropriate first and last name. For instances of addresses, ZIP codes, cities, and states, we found a Python library that randomly returns a real address in the United States, and parsed the output to populate all of the listed fields. When generating these DataFrames, we had to keep in mind some key constraints of our system. First, the arrival and departure airports for a flight had to be distinct. Second, when we booked a trip for a customer, we had to go back and update the seat table to show the seat was booked, and additionally, mark a flight as full if all of the seats were taken. Third, when assigning a flight for a pilot, we had to mark the flight as having an assigned pilot, and not assign another pilot. Finally, when creating any alphanumeric attribute, we had to make sure the attribute was distinct. This was most applicable when generating gate numbers, which are in the form of ‘10C’. While a gate does not need to be unique globally, it must be unique for a particular airport.

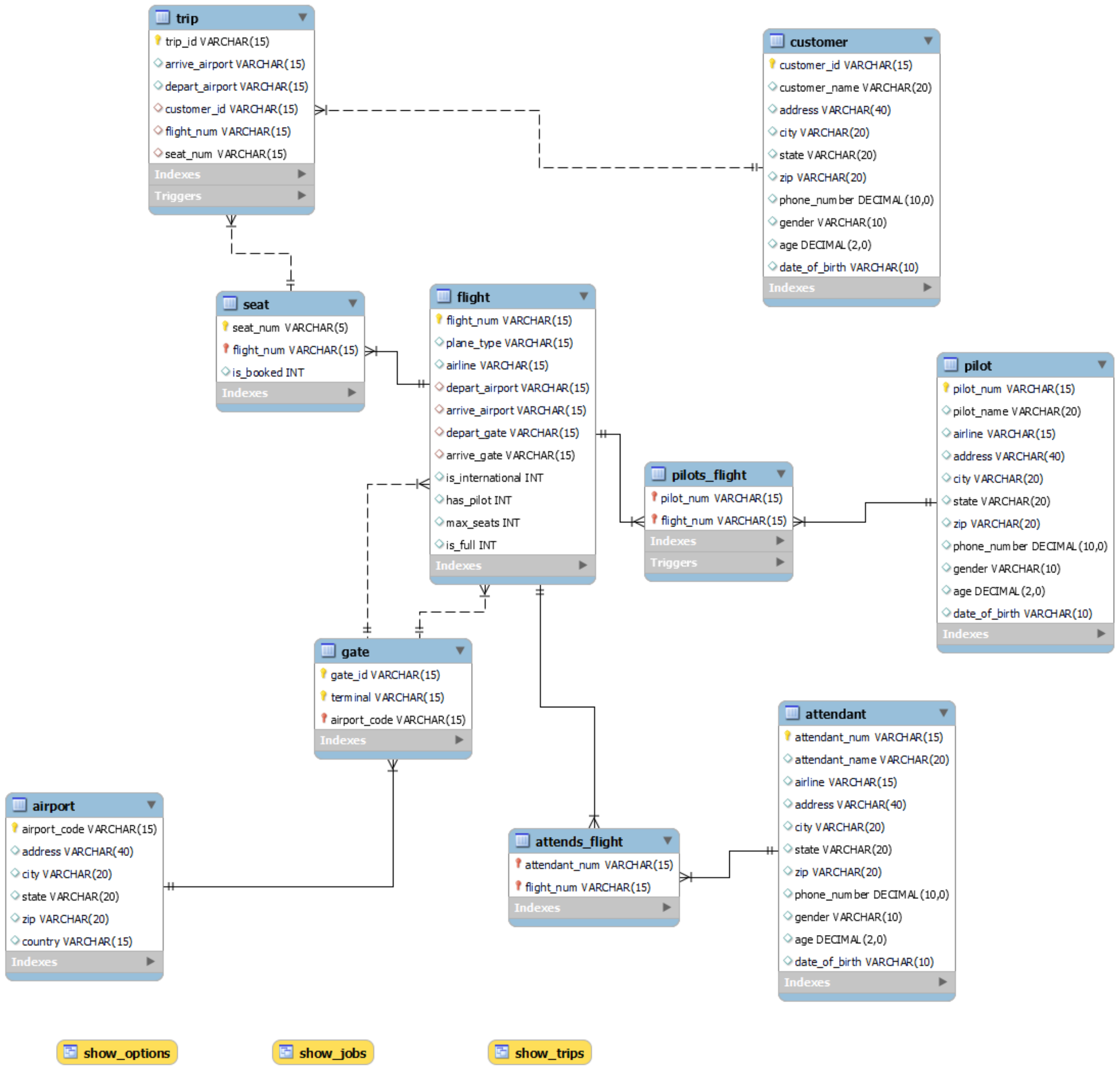
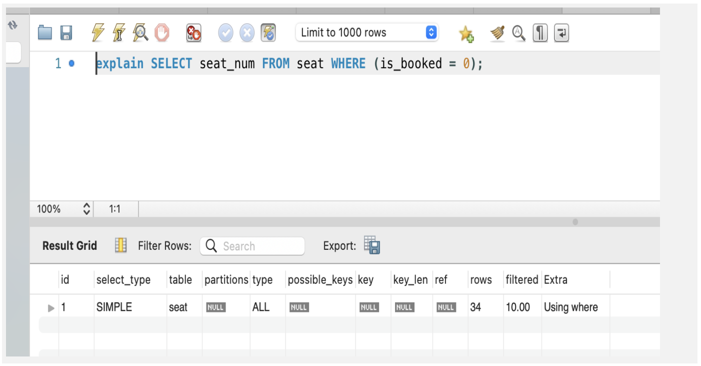


Fig. 1: Entity-Relationship Diagram

Query Optimization

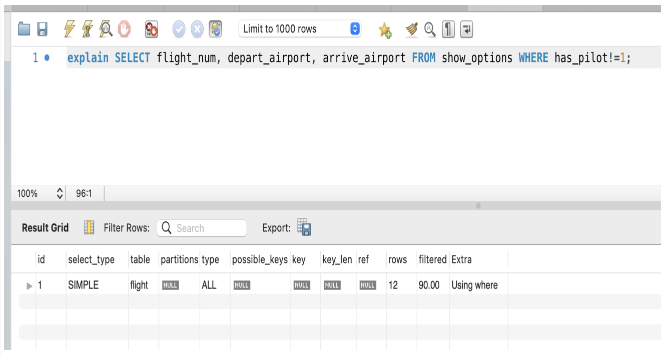
We made use of the 'explain' keyword in MySQL to get information about the query execution plan and how the queries are being executed. Since all of our queries were of 'simple' type, further query optimization was not required.

You should have at least one query with group by / join which can be optimized.



```
explain SELECT seat_num FROM seat WHERE (is_booked = 0);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	seat	NULL	ALL	NULL	NULL	NULL	NULL	34	10.00	Using where



```
explain SELECT flight_num, depart_airport, arrive_airport FROM show_options WHERE has_pilot!=1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	flight	NULL	ALL	NULL	NULL	NULL	NULL	12	90.00	Using where

Application Description

The application is built with the Flask framework and uses MySQL, Python3 for running the backend and HTML and CSS for rendering the frontend user interface. The application has three options for logging in - admin, pilot and customer. The admin can add new flights, delete flights, add new pilots and view statistics about the flight bookings. On the pilot login, one can book a new job on a flight and view jobs that are assigned. Customers can create a new account or choose to sign in with login credentials. On log in, they can book a new flight and select available seats or can view previously booked seats. On booking a ticket, they can also choose to get a pdf print of the ticket.

The data is stored in a MySQL database with different tables for different types of entities like customers, flights, trips, etc. The application makes use of the PyMySQL and MySQL connector libraries for connecting with the database and executing queries from the Python script.

The application uses several different views, procedures, functions, transactions, and triggers to execute different parts of the functionality. First, we have implemented three views. We have a view to show available jobs for a pilot, a view to show booked trips for a customer, and a view to show available flight options to a customer.

There are five stored procedures to help with insert operations. There is one procedure each for adding entries to the customer, pilot, flight, trip, and pilots_flight relations.

There are 3 functions used to display some statistics to an administrator. One function returns the pilot with the most flights scheduled, one returns the most popular destinations booked in trips, and one returns the customer with the greatest number of bookings.

We also have 3 transactions. When inserting data on the customer, pilot, and trip fields, there are a few considerations we had to take into account. First, some fields must be computed before inserting

No need for a transaction if it contains a single DML command

an entry into a table. For example, in order to insert into the customer table, the age must be computed using the current date and the input date of birth. Additionally, if two customers try to book the same seat on the same flight simultaneously, we need the functionality to roll one of the transactions back.

Finally, we have three triggers. First, after insert on the trip relation, we must set seat.is_booked on the booked seat. Second, after insert on trip, we need to check if the flight is full and if so, set flight.is_full correspondingly. Finally, after inserting on the pilots_flight table, we must set flight.has_pilot for the corresponding flight.

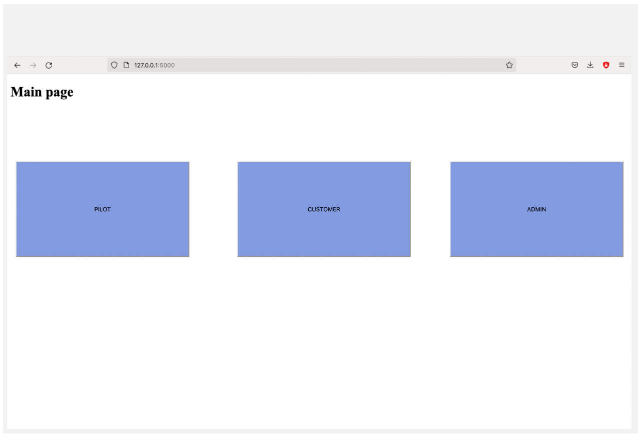


Fig. 2: Main Page

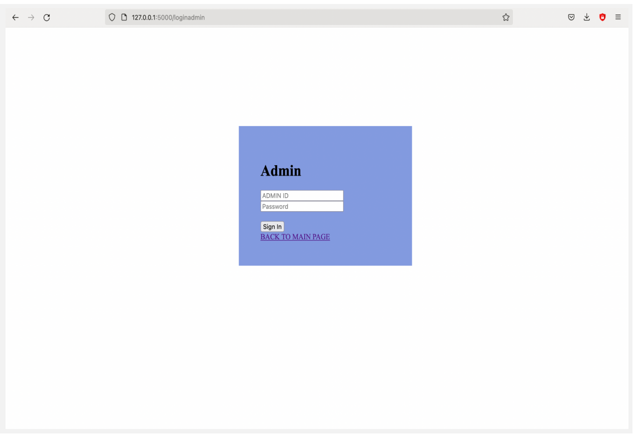


Fig. 3: Admin Login

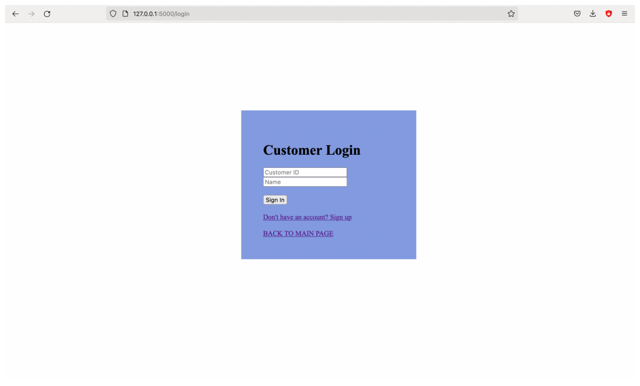


Fig. 4: Customer Login

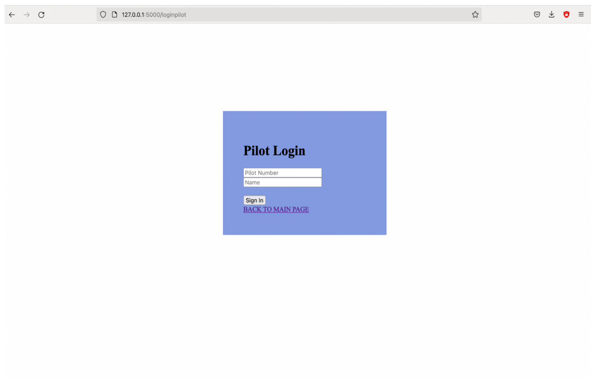


Fig. 5: Pilot Login

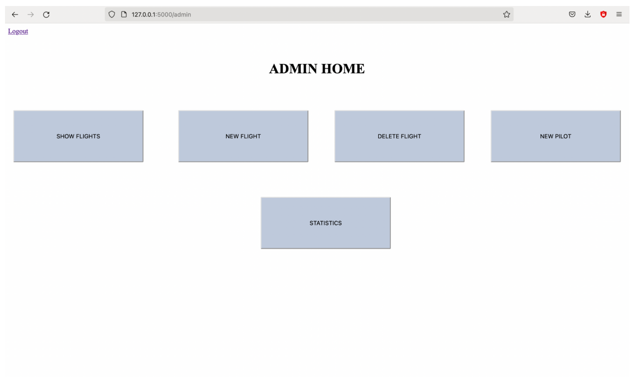


Fig. 6: Admin Home

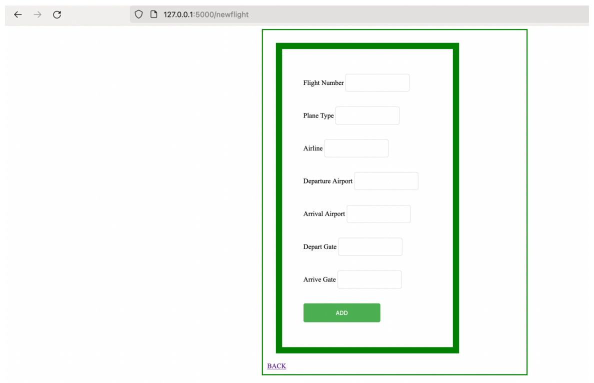
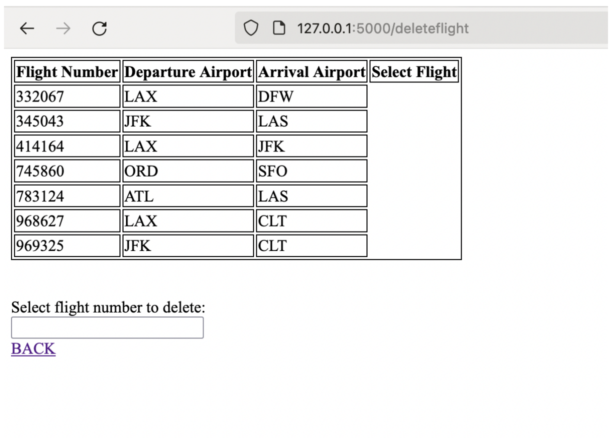


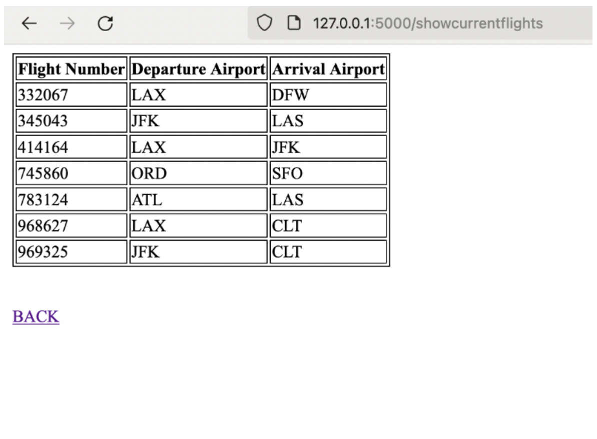
Fig. 7: New Flight


127.0.0.1:5000/deleteflight

Flight Number	Departure Airport	Arrival Airport	Select Flight
332067	LAX	DFW	
345043	JFK	LAS	
414164	LAX	JFK	
745860	ORD	SFO	
783124	ATL	LAS	
968627	LAX	CLT	
969325	JFK	CLT	

Select flight number to delete:

BACK

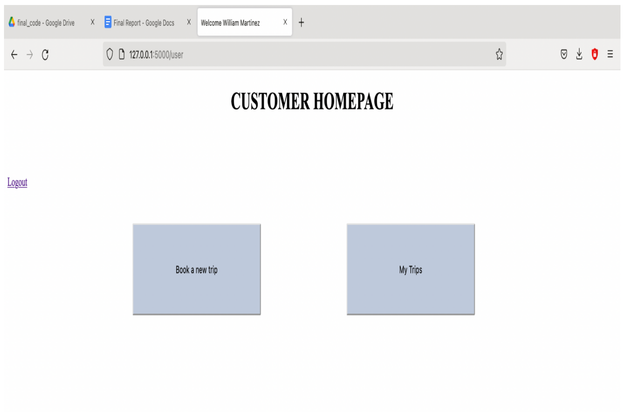

127.0.0.1:5000/showcurrentflights

Flight Number	Departure Airport	Arrival Airport
332067	LAX	DFW
345043	JFK	LAS
414164	LAX	JFK
745860	ORD	SFO
783124	ATL	LAS
968627	LAX	CLT
969325	JFK	CLT

BACK

Fig. 8: Delete Flight

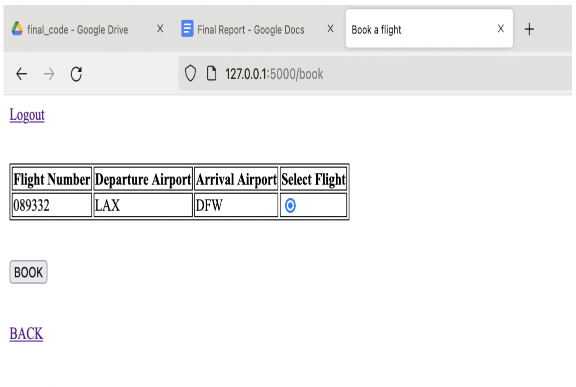
Fig. 9: Show Flights


final_code - Google Drive | Final Report - Google Docs | Welcome William Martinez

127.0.0.1:5000/user

CUSTOMER HOMEPAGE

Logout


final_code - Google Drive | Final Report - Google Docs | Book a flight

127.0.0.1:5000/book

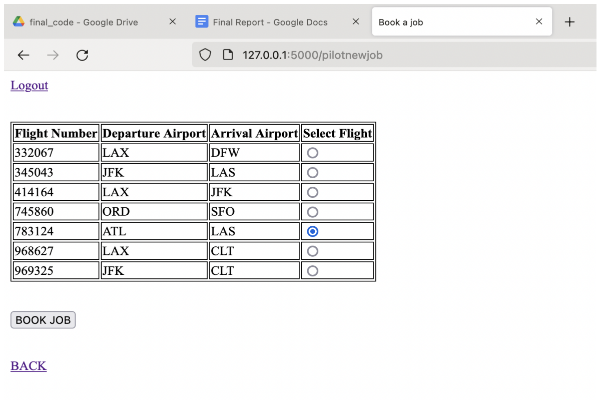
Logout

Flight Number	Departure Airport	Arrival Airport	Select Flight
089332	LAX	DFW	●

BACK

Fig. 10: Customer Home

Fig. 11: Book Flight

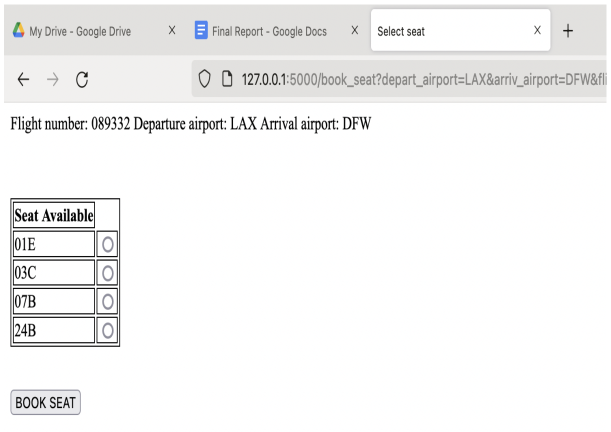

final_code - Google Drive | Final Report - Google Docs | Book a job

127.0.0.1:5000/pilotnewjob

Logout

Flight Number	Departure Airport	Arrival Airport	Select Flight
332067	LAX	DFW	○
345043	JFK	LAS	○
414164	LAX	JFK	○
745860	ORD	SFO	○
783124	ATL	LAS	●
968627	LAX	CLT	○
969325	JFK	CLT	○

BACK


My Drive - Google Drive | Final Report - Google Docs | Select seat

127.0.0.1:5000/book_seat?depart_airport=LAX&arriv_airport=DFW&fl

Flight number: 089332 Departure airport: LAX Arrival airport: DFW

Seat Available	
01E	○
03C	○
07B	○
24B	○

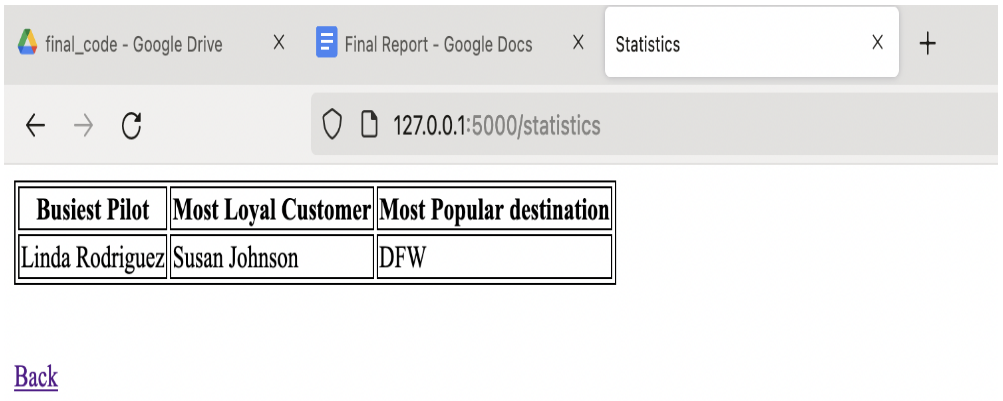
Fig. 12: New Job

Fig. 13: Seat Select

Data Analysis

For performing data analysis on the data, stored functions were used. The functions return the names of the busiest pilot i.e. pilot who has the most flights scheduled, the most loyal customer - the customer who has booked the highest number of trips, and finally the most popular destination which has the highest number of bookings.

We also have some plots generated, including a histogram of airport destinations, shown below.


final_code - Google Drive
Final Report - Google Docs
Statistics
127.0.0.1:5000/statistics

Busiest Pilot	Most Loyal Customer	Most Popular destination
Linda Rodriguez	Susan Johnson	DFW

Back

Fig. 14: Statistics

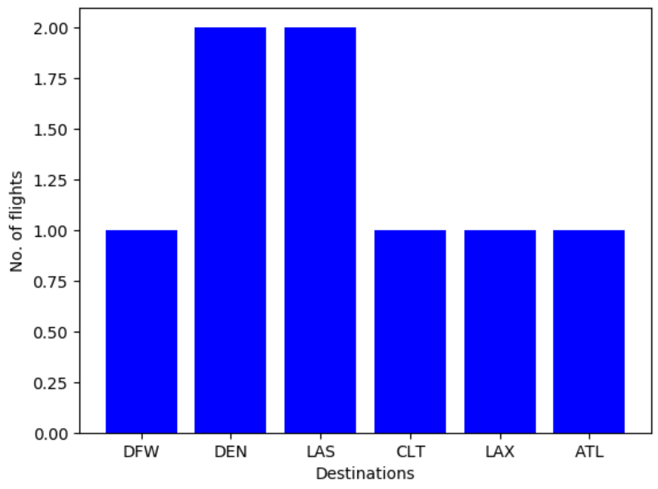


Fig. 15: Statistics

Conclusions

During the course of the project, we learned a lot about design principles. Designing a functional application is much different from creating a one-off database for a homework assignment. Specifically, we learned that we need to be careful with updating other relations when making changes to one relation. This came up several times with our trips table. This led to us adding some triggers on an insert to this table, including updating the booked seat and checking whether the flight is full.

Given more time for this project, we would have made several changes and increased the project scope. The first change we would make is adding another relation to allow a trip to include multiple legs. As an example, if a customer wants to fly from BOS to LAX, they could either fly direct, or take two flights from BOS to DEN, and DEN to LAX. This was originally part of our design, but it was simplified in order to meet the deadlines.

Second, we would add some more detail to our Flask application. This would involve adding some CSS and JavaScript to make our website look better. Though this wouldn't necessarily improve the functionality, it would provide a better experience to the end user. We would additionally add website banners, custom logos, and loading screens.

Third, we would add some pricing aspects to our system. We would make it so that each flight is separated into first class, business, and economy, with different ticket costs for each section. There could also be a component with adjustable prices for a window or aisle seat, and extra cost for seats with extra leg room. There would also be an option to check bags, sign up for early boarding, and other convenience features. These factors would all add up to the final price of a ticket.

After completing this project, we have some advice for future students. Most importantly, we would advise not overcomplicating the scope. Our original design was very complicated, and we eventually had to make compromises for the sake of deadlines, removing from the total scope. A better way to do it might be to define a small set of necessary functionality, and several levels of additional features to add if time permits.