Team 3, Target 1

William Berquist
Paul Edson

Installations

This team’s web application used the following installations:

- npm install node
- npm install node-fetch
- npm install cors
- npm install express

Structural Format

The web application's queries and transactions are stored in the index.js file. We get user input from the inputForm.html. inputForm.html calls testIndex.js which then loads up all of our javascript. Information is sent to the file exported.js for client side processing. Validation is done, information is saved in an array of dictionaries, and that array is send to the file databaseFunctions.js, which stringifies the object and then sends it to the backend called index.js. Index.js turns it back into an array of dictionaries, prepares the information to be sent to the database, and then sends it to the database. The results of the query / transactions are sent back to the user in the reverse order.

The flow works like this:

inputForm.html ⇒ testIndex.js ⇒ exported.js ⇒ databaseFunctions.js ⇒ index.js ⇒ databaseFunctions ⇒ exported.js ⇒ testIndex.js ⇒ inputForm.html

Input Guide

"User Name" on inputForm.html is referred to as SSN in the backend and frontend code. It is only called User Name on the webpage itself. A username can be any letter or number as long as nobody else has registered that username yet.

A credit card number can be any length of numbers to be added. We figured that if this were real, we would have an actual payment process that checks it. Regardless, we did not have time to add this validation.

The maximum number of bags that can be taken on a plane is 4.

There are only a few valid discount codes:

- 10PCTOFF, 20PCTOFF, 10DOLOFF, 20DOLOFF, SEASONAL10PCT, SEASONAL20DOL

Searching for a flight requires that the first letter of each city is capitalized,

- ie: Houston, not houston. New York, not new york and not New york.

Current valid cities that have flights:

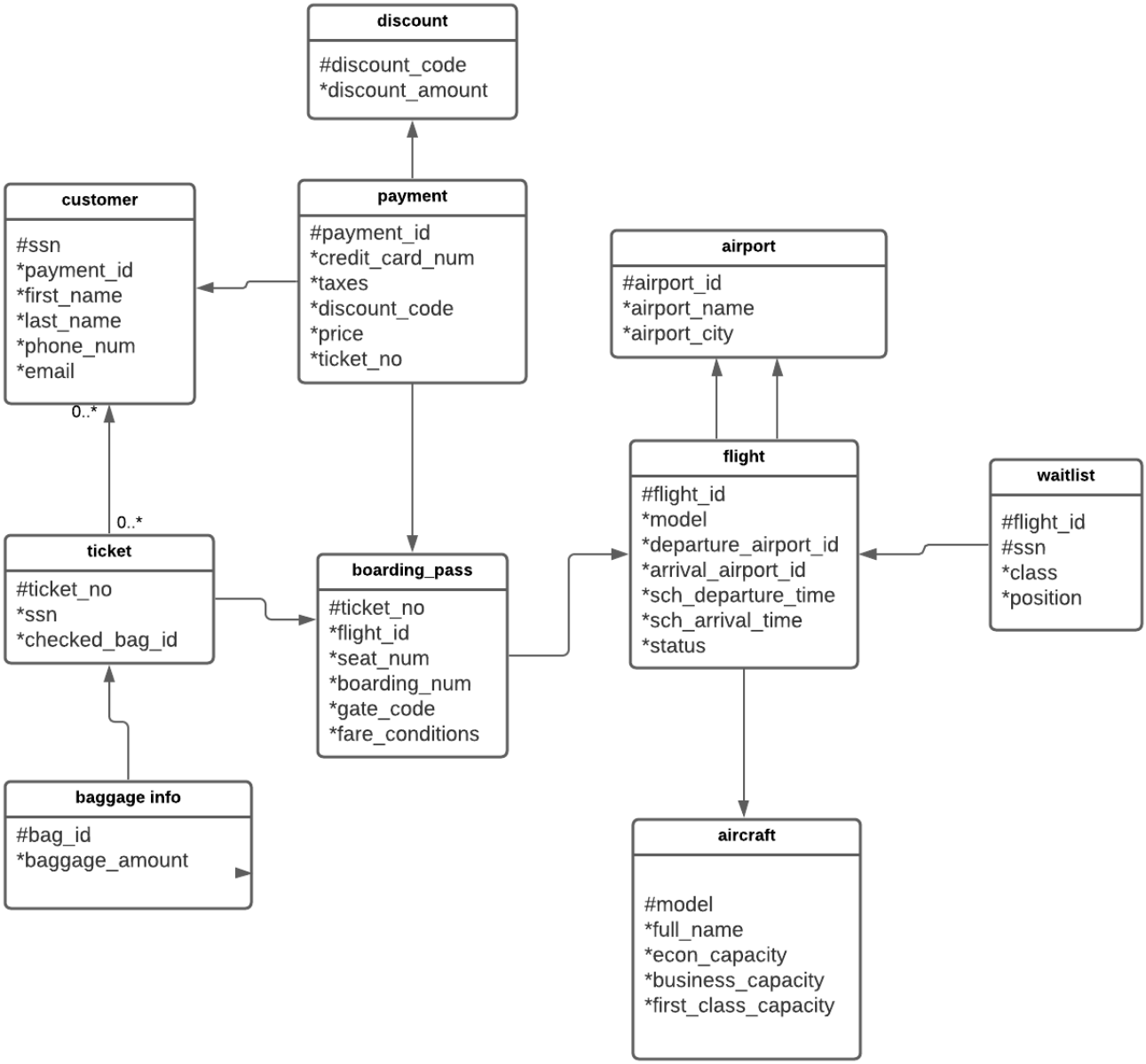
- Denver
- Dallas
- New York
- Austin
- Los Angeles
- Houston

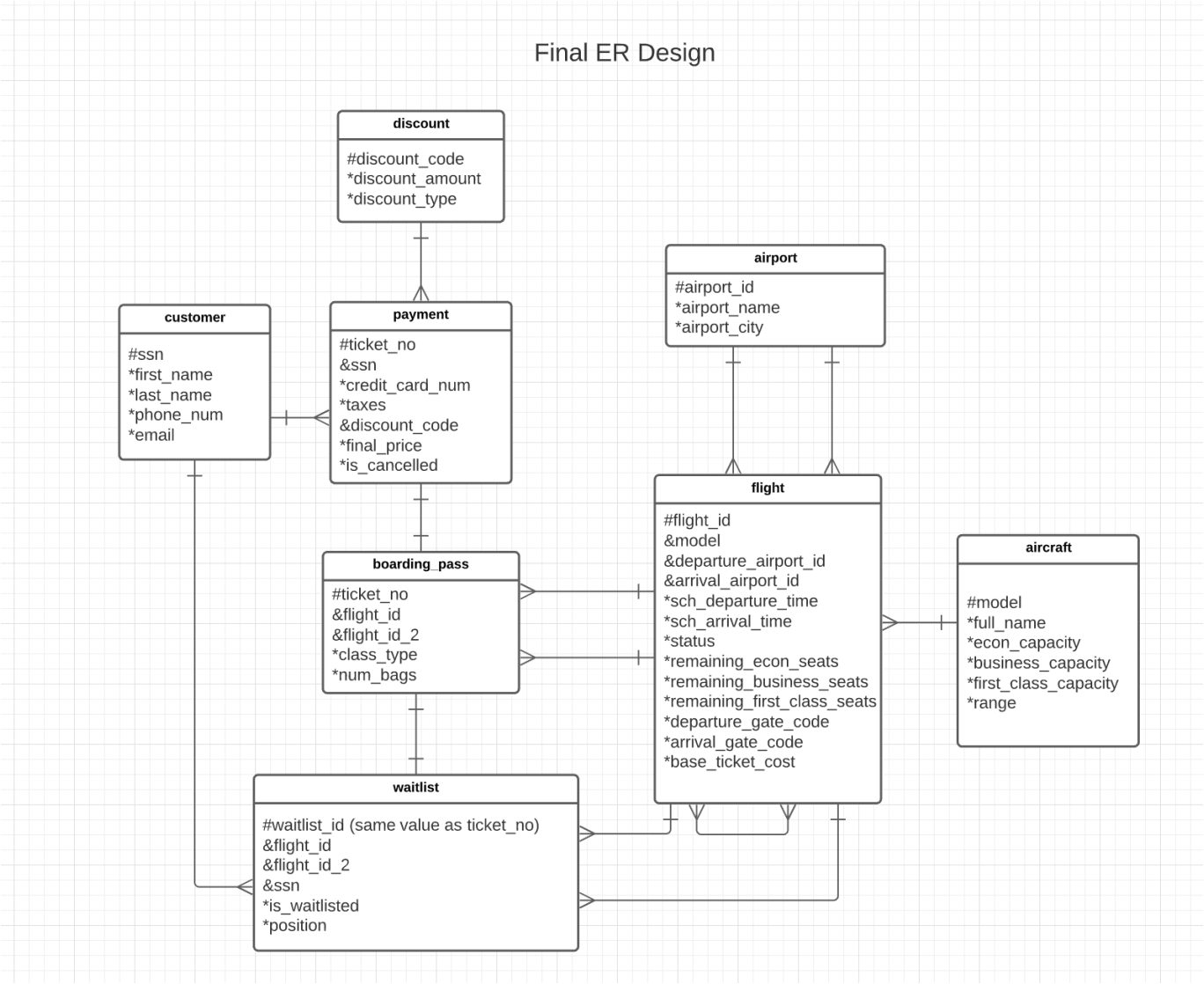
Each of these cities have multiple flights to and from.

There is a base ticket cost, which is the cost of an economy seat. a business seat costs 3 times the base ticket cost, and first class costs 5 times the base ticket cost.

The best way to test the waitlist is to use flight ID = 2 with class type business or first. These only have 1 business and 1 first class seat each, so you can quickly fill up the flight and then test it out. If you want to check the waitlist for a connecting flight, use flight ID = 2 and flight ID 2 = 3.

Initial ER Design





ER diagram notes

symbol means primary key

& means primary key

* means either a non-key column

There are no dotted lines as NULLs are not allowed at all in the database. Instead we use ‘NA’ or -1 for fields that are empty. In general using -1 can be dangerous, but there is never a time when we use negative numbers on our database, so it works well enough for our purposes.

We allow a customer to buy as many tickets for a flight as they want. If they want extra room on the flight, they can buy more tickets. If they want to throw a party on a flight, they can buy every seat on the flight. This means that we are required to use an auto generated PK for a few tables, but we want to keep our customers happy.

We do not have a specific bag_id. Instead all bags are referenced from the ticket_no.

ER Diagram: Normalization

In this section we list the table name along with the pk(s). We then say what normal form it is in, and make clear why certain columns that look like they might break normalization do not actually break it. We also give justifications for auto-incrementing PKs and for tables that are not in 3NF or BNCF.

Customer table:

- PK is ssn. The table is in BCNF.
    - Both email and phone number are optional columns for the customer to give, and are filled in as 'NA' if left blank.

Payment table:

- PK is ticket_no. The table is in BCNF.
- ssn is a FK from the customer table
- ticket_no is a FK from the boarding_pass table.
- discount_code is a FK from the discount table.
    - It is possible for there to be multiple ticket_no and multiple credit_card_num for a given ssn.

Boarding_pass table:

- PK is ticket_no. The table is in BCNF
- flight_id and flight_id_2 are FKs from the flight table
- waitlist_id is a FK from the waitlist table
    - ticket_no is autogenerated by the DBMS. This is justified as there is no other column that would be unique, and it is a fitting PK for a boarding pass. Even if we added ssn to this table, it is possible for a customer to reserve multiple seats on a single flight under their name. Therefore ssn, flight_id, and flight_id_2 as a composite key would not be guaranteed to be unique.

Discount table:

- PK is discount_code. The table is in BCNF.
    - There are multiple codes that give the same discount_amount, and there are only two discount_type, 'Percent' and 'Dollar'.

Airport table:

- PK is airport_id, which is the 3 letter code for an airport. The table is in 3NF only.
    - This is because airport_name → airport_code. This is justified because the table is very small. It would be worse to split it up even more to make it fully normalized, as this would require more joins.

Flight table:

- PK is flight_id. This is an auto generated code from the DBMS. The table is in BCNF.
- departure_airport_id and arrival_airport_id are both FKs from the airport table.
- model is a FK from the aircraft table

Aircraft table:

- PK is model, which is a 3 letter code. This table is in 3NF, as full_name → model. (full name means the full name of the plane).

Waitlist table:

- PK is waitlist_id, which is the same value as ticket_no when someone is added to the waitlist.
    - Adding someone to the waitlist adds their information to boarding_pass and payment table as well, with the column is_waitlisted set to TRUE. This table is in 3NF, as position → waitlist_id. position is an auto-incremented integer. The reasons for this are explained elsewhere. Even though this table is in 3NF, having position work this way greatly improves efficiency when people are removed from the waitlist. This is also explained elsewhere.

Important Queries

Query to find connected flights:

```
SELECT
t1.flight_id AS flight_id1,
t2.flight_id AS flight_id2, t1.departure_airport_id,
t1.arrival_airport_id AS layover_airport_id,
t2.arrival_airport_id AS destination_airport_id,
t1.sch_departure_time, t1.sch_arrival_time AS layover_arrival_time,
t2.sch_departure_time AS layover_departure_time,
t2.sch_arrival_time AS destination_arrival_time,
t1.economy_seat_left AS t1_econ,
t2.economy_seat_left AS t2_econ, t1.business_seat_left AS t1_bus
t2.business_seat_left AS t2_bus,
t1.first_class_seat_left AS t1_first,
t2.first_class_seat_left AS t2_first

FROM flight AS t1

INNER JOIN flight AS t2 ON t2.departure_airport_id = t1.arrival_airport_id

WHERE t1.departure_airport_id =
(SELECT airport_id FROM airport WHERE airport_city = '${r.rows[0].airport_city}')

AND t2.arrival_airport_id =
(SELECT airport_id FROM airport WHERE airport_city = '${s.rows[0].airport_city}')

AND t1.sch_arrival_time < t2.sch_departure_time

AND t1.flight_id = ${f.flightID} AND t2.flight_id = ${f.flightID2};
```

This query joins the flight table onto itself to find connected flights between cities. Joining the departure airport of one flight to the arrival airport of another flight gives all possible connected flights between any two cities, not accounting for time constraints. The where clause trims down this table to only the relevant flights by removing the connected flights that don’t start in the departure airport and arrive in the arrival airport input by the user. The where clause then also removes connected flights where the first flight lands after the second flight has already taken off. This leaves a table of connected flights that only include relevant flights that account for time constraints.

Transaction to buy a ticket

```
let query = `BEGIN TRANSACTION;

CREATE TEMP TABLE boughtTicks(
ticketNo INT,
finalPrice FLOAT,
ssn VARCHAR(50),
flightID VARCHAR(50),
flightID2 VARCHAR(50)
);

for(i = 0; i < t.length; i++)
{
    query = query +

    WITH ins${i} AS (
    INSERT INTO boarding_pass (
        flight_id, flight_id_2, class_type, num_bags, is_waitlisted)
    VALUES (
        ${t[i].flightID}, ${t[i].flightID2},
        '${t[i].classType}', ${t[i].numBags}, 'FALSE')
    RETURNING ticket_no)


    INSERT INTO payment (
        ticket_no, ssn, credit_card_num, taxes, discount_code,
        final_price, flight_id, flight_id_2,is_cancelled)
    VALUES (
        (SELECT ticket_no FROM ins${i}), '${t[i].ssn}',
        '${t[i].creditCardNum}', 'NA',
        '${t[i].discountCode}', ${t[i].totalCost},
        ${t[i].flightID}, ${t[i].flightID2}, FALSE);


    INSERT INTO boughtTicks(
        ticketNo, finalPrice, ssn, flightID, flightID2)
    VALUES (
        (SELECT ticket_no FROM payment ORDER BY ticket_no DESC limit 1),
        ${t[i].totalCost}, ${t[i].ssn}, ${t[i].flightID}, ${t[i].flightID2});


    UPDATE flight
```

```
    SET ${seatsLeftDict[t[i].classType]} = (SELECT ${seatsLeftDict[t[i].classType]}
    FROM flight WHERE flight_id = ${t[i].flightID}) - 1
    WHERE flight_id = ${t[i].flightID};



    UPDATE flight
    SET ${seatsLeftDict[t[i].classType]} =
        CASE ${t[i].flightID2}
            WHEN -1 THEN 0
            ELSE (SELECT ${seatsLeftDict[t[i].classType]}
                FROM flight WHERE flight_id = ${t[i].flightID2}) - 1
        END
    WHERE flight_id = ${t[i].flightID2};`

}
query = query + `END TRANSACTION;`
```

Getting information to the backend
When a customer wants to buy between 1 and 5 tickets at once, we use this transaction to do it.

We take in the customers SSN, the ID of the flight they want to book, an optional second flight ID for a connecting flight, an optional discount code, their credit card number, and the number of bags they want to bring. We do validation on this information, such as checking to see if it is a valid flight ID, if it's a registered SSN, if the discount code is valid, and if the connecting flight ID is actually a valid connecting flight.

If these validation checks are passed, then we add this data to a variable which is an array of dictionaries, with each element of the array being the information for a single ticket. Then this is passed to our backend code.

We use this variable to build a string of a very large transaction, using a loop with the number of tickets to be bought as the index. Once the string is built, we send it to the database to be processed.

How the query works
We start by creating a temporary table boughTicks. The information on this table is what we will return to the frontend to display to the customer. Then we enter our loop to create the proper insertions and updates for each ticket.

The first thing we do is create a boarding pass on the boarding_pass table for this ticket. The ticket_no is autogenerated by the database, and we want to keep this information to use, so we use the WITH ... RETURNING clause to achieve this.

Then we add the customer's payment information to the payment table. We take ticket_no we just made and add it to this table as the primary key.

Next we insert the information we want to return to the database in the boughtTicks temporary table.

The last thing we want to do is update the flight table to decrease the number of tickets available for the given class type (economy, business, etc). Since It is possible to have two flight IDs, we will need to possibly update two flights. This is why we use the conditional in the second update statement. We do nothing if the second flight ID is NA (if a customer leaves the second flight field blank when buying a ticket, the frontend changes this value to 'NA' so we don't have to manage nulls on the database). Otherwise we also decrease the second flight's number of seats for the particular class by 1.

After doing this process for each ticket, we exit the loop, add the final END TRANSACTION statement to the string, and send the string to the database. We do a bit more validation based on the results of the transaction, such as there not being enough seats left. We then return the result to the customer on the frontend, either by telling them that not enough space was left, or by telling them they successfully bought the ticket and giving them their ticket number.

Query to gather ticket information for the customer

```
SELECT
boarding_pass.ticket_no, boarding_pass.flight_id, boarding_pass.flight_id_2,
boarding_pass.class_type, boarding_pass.num_bags, waitlist.is_waitlisted,
payment.final_price, payment.is_cancelled,
f1.departure_airport_id AS f1_departure_airport_id,
f1.arrival_airport_id AS f1_arrival_airport_id,
f2.departure_airport_id AS f2_departure_airport_id,
f2.arrival_airport_id AS f2_arrival_airport_id,
f1.sch_departure_time AS f1_sch_departure_time,
f1.sch_arrival_time AS f1_sch_arrival_time,
f1.status AS f1_status,
f1.departure_gate_code AS f1_departure_gate_code,
f1.arrival_gate_code AS f1_arrival_gate_code,
f2.departure_airport_id AS f2_departure_airport_id,
f2.arrival_airport_id AS f2_arrival_airport_id,
f2.sch_departure_time AS f2_sch_departure_time,
f2.sch_arrival_time AS f2_sch_arrival_time,
f2.status AS f2_status,
f2.departure_gate_code AS f2_departure_gate_code,
f2.arrival_gate_code AS f2_arrival_gate_code,
depAirport.airport_name AS f1_dep_airport_name,
depAirport.airport_city AS f1_dep_airport_city_name,
arAirport.airport_name AS f1_arr_airport_name,
arAirport.airport_city AS f1_arr_airport_city_name,
arFinalAirport.airport_name AS f2_arr_airport_name,
arFinalAirport.airport_city AS f2_arr_airport_ciy_name

FROM boarding_pass

INNER JOIN payment ON boarding_pass.ticket_no = payment.ticket_no
INNER JOIN flight f1 ON boarding_pass.flight_id = f1.flight_id
INNER JOIN flight f2 ON boarding_pass.flight_id_2 = f2.flight_id
INNER JOIN airport depAirport ON f1.departure_airport_id = depAirport.airport_id
INNER JOIN airport arAirport ON f1.arrival_airport_id = arAirport.airport_id
INNER JOIN airport arFinalAirport ON f2.arrival_airport_id =
arFinalAirport.airport_id
LEFT JOIN waitlist ON boarding_pass.ticket_no = waitlist.waitlist_id

WHERE payment.ticket_no = ${t.ticket_no} AND payment.ssn = '${t.ssn}';
```

This is a very large query to gather information. If a customer wants to get information about their ticket they have bought, there is relevant information on many tables that is needed. We need to access the boarding_pass table, the payment table, the flight table, and the airport table. We need to access the airport table 3 times because there may be a connecting flight (departure airport, layover airport, and arrival airport). We also need to access the flight table 2 times because there may be two flights. This is why there are 6 joins. If there is no connecting flight, there will be 'NA' returned for many of the things we are selecting, which will get taken care of in the frontend.

Transaction to add someone to the waitlist

```
BEGIN TRANSACTION;

CREATE TEMP TABLE waitlistInfo(
waitlist_id INT,
ssn VARCHAR(50),
flightID VARCHAR(50),
flightID2 VARCHAR(50),
position INT
);

WITH ins0 AS (
    INSERT INTO boarding_pass
        (flight_id, flight_id_2, class_type, num_bags)
    VALUES
        (${w.flightID}, ${w.flightID2}, '${w.classType}', ${w.numBags})
RETURNING ticket_no)

INSERT INTO payment
    (ticket_no, ssn, credit_card_num, taxes,
    discount_code, final_price, is_cancelled)
VALUES
    ((SELECT ticket_no FROM ins0), '${w.ssn}', '${w.creditCardNum}',
    'NA', '${w.discountCode}', ${w.totalCost}, FALSE);

INSERT INTO waitlist
    (waitlist_id, ssn, flight_id, flight_id_2, is_waitlisted)
VALUES
    ((SELECT ticket_no FROM payment ORDER BY payment DESC limit 1),
    ${w.ssn}, ${w.flightID}, ${w.flightID2}, 'TRUE');

INSERT INTO waitlistInfo
    (waitlist_id, ssn, flightID, flightID2, position)
VALUES
    ((SELECT waitlist_id FROM waitlist ORDER BY waitlist_id DESC limit 1),
    ${w.ssn}, ${w.flightID}, ${w.flightID2},
    (SELECT position FROM waitlist ORDER BY waitlist_id DESC limit 1));

END TRANSACTION;
```

If a customer wants to be added to the waitlist for a flight, they can enter their information in the waitlist form to do so. The information they enter is identical to what they would enter if

they wanted to buy a single ticket. This is because we will be adding their information to 3 tables: waitlist, boarding_pass, and payment.

We are doing this because if a spot opens up for them, we want to instantly put them on a flight. This means accessing the boarding_pass and payment tables to make it an official ticket. There is a column on the boarding_pass table called is_waitlisted, which is set to TRUE whenever someone is on a waitlist and is automatically set to false whenever someone buys a ticket or is moved off the waitlist.

The first thing we do when someone wants to be added to the waitlist is validate their data (SSN exists on database, flight exists, etc). Then we run a query to check whether the flight they want to waitlist on is actually full. If there are still seats left, we tell them and don't let them waitlist. If the flight is full, we begin the transaction to add them to the waitlist.

Next we create a temporary table that will be filled with information we want to return to the customer once we add them to the waitlist. Then we add their information to the boarding_pass table, the payment table, and the waitlist table.

Relevant information such as waitlist_id and position are added to the temporary table and then the temporary table is returned to the frontend. We will need to do one last query though, which is using the position value to calculate the actual position of the customer.

Query to check the position of a customer on the waitlist

```
SELECT position
FROM waitlist
WHERE position <= ${p.position} AND is_waitlisted = 'TRUE'
AND flight_id = ${p.flightid} AND flight_id_2 = ${p.flightid2}
ORDER BY position ASC;
```

The position column on the waitlist table is a bit misleading. It is an auto incrementing column that is not unique for a particular flight. Eg: adding a person from flight A will have the position of X, and then if the next person waitlisted is on flight B, the position value will be X + 1. While this seems strange, it actually allows us to calculate a customer's position on the waitlist much easier, and allows us to keep track of every customers position when someone is removed from the waitlist *much* more efficiently.

What we do is select all of the records on the waitlist of people who are on a particular flight, in the customers class, and have the is_waitlisted field set to TRUE. We order this in ascending order and filter it by records of customers whose position is less than or equal to the particular customer's position value we are trying to find. This means that the customer's position we are trying to calculate will be the very last record of the records returned. Therefore the rowCount of the query results will be the customer's position on the waitlist.

Thus we can just return the rowCount of the query. This may seem convoluted, but it has a big advantage: we never have to change the waitlist position on the database whenever someone is taken off the waitlist. If we did, we would have to update every single person's waitlist record for a particular flight. That is very inefficient, so we found another way to do it.

Since we look for the flag is_waitlisted, this will always work. If someone is moved off the waitlist, the rowCount returned would be one less.

Transaction for cancelling tickets

```
// checks if there is anyone on the waitlist of the flight(s)
// and class of the person cancelling their ticket

s = await pool.query(

    SELECT *
    FROM waitlist AS w
    INNER JOIN boarding_pass AS b ON waitlist_id = ticket_no
    WHERE w.flight_id = ${r.rows[0].flight_id}
        AND w.flight_id_2 = ${r.rows[0].flight_id_2}
        AND w.is_waitlisted = 'TRUE' AND b.class_type = '${r.rows[0].class_type}';`
    );

// nobody was on the waitlist, so just cancel the ticket
if(s.rowCount === 0) {
    q = await pool.query (

    BEGIN TRANSACTION;

        UPDATE payment
        SET is_cancelled = true
        WHERE ticket_no = ${t.ticket_no};

        UPDATE flight
        SET ${seatsLeftDict[t.classType]} = ${seatsLeftDict[t.classType]} + 1
        WHERE flight_id =
            (SELECT flight_id FROM boarding_pass WHERE ticket_no = ${t.ticket_no});

    END TRANSACTION;

    );

// someone was on the waitlist, so cancel the ticket,
// and move the first person on the waitlist onto the flight
} else {

    w = await pool.query(

BEGIN TRANSACTION;

UPDATE payment
```

```
SET is_cancelled = TRUE
WHERE ticket_no = ${t.ticket_no};

UPDATE waitlist
SET is_waitlisted = FALSE
WHERE waitlist_id = (
    SELECT waitlist_id FROM (
        SELECT *
        FROM waitlist AS w
        INNER JOIN boarding_pass AS b ON waitlist_id = ticket_no
        WHERE w.flight_id = ${r.rows[0].flight_id}
        AND w.flight_id_2 = ${r.rows[0].flight_id_2}
        AND w.is_waitlisted = 'TRUE'
        AND b.class_type = '${t.classType}') AS activeWaitlist
        ORDER BY activeWaitlist.position ASC limit 1);

END TRANSACTION;`
```

First we get the ticket_no and SSN of someone who wants to cancel their ticket. We do some validation, such as checking if the ticket has already been cancelled. Then we query the waitlist table to see if there is anyone on the waitlist for the flight of the person cancelling their ticket with the same class.

If nobody is on the waitlist, we simply do two updates to the database. First we update the is_cancelled value from FALSE to TRUE, and then we update the number of seats remaining of the class on the flight of the cancelled ticket.

If there is someone on the waitlist, we also set the is_cancelled field to TRUE, but we do not update the number of seats on the flight. Instead we find the first person on the waitlist for the flight using a similar query above, and then change their is_waitlisted status to false.**