Refer to View Public Info in Home page when not logged in for the general layout of my queries when I look for flight data

1. Home page when not logged-in:
    a. View Public Info (publicRouter.mjs)
        i.
```
`SELECT
  Flight.*,
  dep_airport.city AS dep_city,
  dep_airport.port_name AS dep_name,
  arr_airport.city AS arr_city,
  arr_airport.port_name AS arr_name
FROM Flight
JOIN Airport AS dep_airport ON Flight.dep_port = dep_airport.port_code
JOIN Airport AS arr_airport ON Flight.arr_port = arr_airport.port_code
`;
```
        ii.
```
//swap fields depending on if the user is wants to depart from or arrive from a location
let srcCityField   = depArr === "Arrival" ? "arr_airport.city" : "dep_airport.city";
let srcNameField   = depArr === "Arrival" ? "arr_airport.port_name" : "dep_airport.port_name";
let destCityField  = depArr === "Arrival" ? "dep_airport.city" : "arr_airport.city";
let destNameField  = depArr === "Arrival" ? "dep_airport.port_name" : "arr_airport.port_name";

//If the user provides any of the following fields, add them to the conditions and values list
if(sourceCity) {
  conditions.push(`${srcCityField} = ?`);
  values.push(sourceCity);
}
if(sourceName) {
  conditions.push(`${srcNameField} = ?`);
  values.push(sourceName);
}
if(destCity) {
  conditions.push(`${destCityField} = ?`);
  values.push(destCity);
}
if(destName) {
  conditions.push(`${destNameField} = ?`);
  values.push(destName);
}
if(date) {
  conditions.push("Flight.dep_date >= ?");
  values.push(date);
}

//If the user added no conditions, I will show them all the flights. Otherwise, I"m going to us
if(conditions.length !== 0){
  query += "WHERE " + conditions.join(" AND ")
}

//Lastly, Order by earliest date to latest date
query += " ORDER BY dep_date ASC";
```
        iii. First I join the flight table with the airport table so that I get access to the port names and cities
        iv. Then, depending on if the user provided any values like city/airport name, destination city/airport name, departure date, returning trip or departure, those conditions are added at the end with a series of where conditions
            1. If the user picks Arrival, it shows trips from the source to dest

2. If the user picks departure, it shows trips from dest to source.
3. So the city and port names are swapped

v. Lastly, we order by the departure date in ascending order

b. Register (authorizationRoutes.mjs)

i.
```
//Selects all staff with email same as the users registered email
connection.query('SELECT * FROM airplaneSystem.airline_staff WHERE username = ?', [username], async (err, results) => {
  if (results.length > 0) {
    res.render('staffRegister', { error: 'Staff already exists' });
  }
  else{
    connection.query('SELECT * FROM airplaneSystem.airline WHERE line_name = ?', [line_name], async (err, results) => {
      if (results.length < 1) {
        res.render('staffRegister', { error: 'Airline does not exist' });
      }
      else{

        let hashedPassword = await bcrypt.hash(password, 10);

        let values = [username, hashedPassword, first_name, last_name, date_of_birth, line_name];

        let query = "INSERT INTO airplaneSystem.airline_staff (username, password, first_name, last_name, date_of_birth, lin
          connection.query(query, values, (err, result) => {
            if(err){
              res.render('staffRegister', { error: err.message });
            }
            req.session.username = username;
            req.session.first_name = first_name;
            req.session.last_name = last_name;
            req.session.date_of_birth = date_of_birth;
            req.session.line_name = line_name;
              res.redirect('/staffHome');
            });
```

ii. First I query the staff table with the provided username to see if the staff member already exists

iii. Then I query the airline table to see if the airline the staff is trying to register for exists

iv. Then I insert into the airline table details the staff provide

v. Then a session is created and they go to their respective homepage

vi. Similarly, I do the same kind of process for customer registration

c. Login (authorizationRoutes.mjs)

i.
```
router.post('/staffLogin', (req, res) => {
    let username = req.body.username;
    let password = req.body.password;
  connection.query('SELECT * FROM airplaneSystem.airline_staff WHERE username = ?', [username], async (err, results) => {
    if (results.length > 0) {
      let found = await bcrypt.compare(password, results[0].password);

      if(found){
        req.session.username = results[0].username;
        req.session.first_name = results[0].first_name;
        req.session.last_name = results[0].last_name;
        req.session.date_of_birth = results[0].date_of_birth;
        req.session.line_name = results[0].line_name;
        res.redirect('/staffHome');
      }
      else{
        res.render('staffLogin', { error: 'Invalid credentials' });
      }
    } else {
      res.render('staffLogin', { error: 'Invalid credentials' });
    }
  });
```

ii. I query to see if the username exists in the table, then I check if the password they provided matches the hashed password returned by the query if the username exists.

iii. Then a session is created and they go to their respective homepage.

iv. This process is similar for both staff and customers

2. Customer Use Cases (customerRouter.mjs)
    a. View My flights
        i.
```
let query =
`SELECT
  Flight.*,
  dep_airport.city AS dep_city,
  dep_airport.port_name AS dep_name,
  arr_airport.city AS arr_city,
  arr_airport.port_name AS arr_name
FROM Ticket
JOIN Flight ON Ticket.flight_num = Flight.flight_num
  AND Ticket.dep_date = Flight.dep_date
  AND Ticket.dep_time = Flight.dep_time
  AND Ticket.line_name = Flight.line_name
JOIN Airport AS dep_airport ON Flight.dep_port = dep_airport.port_code
JOIN Airport AS arr_airport ON Flight.arr_port = arr_airport.port_code
WHERE Ticket.email = ?`;
```
        ii.
```
if(startDate) {
  conditions.push("Flight.dep_date >= ?");
  values.push(startDate);
}
if(endDate) {
  conditions.push("Flight.dep_date <= ?");
  values.push(endDate);
}
```
        iii. Almost exactly the same as View Public Info except I also joined that query with the ticket table where the email of that ticket matches the user's email.
        iv. This way only flights the user has tickets for will show up
        v. Also the user can search with within a range of dates for departure instead of just starting date
    b. Search for flights
        i.
```
let today = new Date().toISOString().split("T")[0];
conditions.push("Flight.dep_date >= ?");
values.push(today);
```
        ii.
```
if(endDate) {
  conditions.push("Flight.dep_date <= ?");
  values.push(endDate);
}
```
        iii. Almost exactly the same as View Public Info except the customer can't submit a starting date and they can specify an ending date also.
        iv. So there is now a where condition that checks if the departure date is greater than or equal to the current date and less than or equal to the specified ending date. This will ensure only future flights are shown
    c. Purchase Tickets

i.
```
let query = `
  SELECT Airplane.seats FROM Flight
  JoIN Airplane ON Flight.plane_ID = Airplane.plane_ID
  AND Flight.airplane_line_name = Airplane.line_name
  WHERE Flight.flight_num = ?
  AND Flight.dep_date = ?
  AND Flight.dep_time = ?
  AND Flight.line_name = ?;
`;
```

ii.
```
let query = `
  SELECT COUNT(*) AS ticket_count FROM Ticket
  WHERE flight_num = ?
  AND dep_date = ?
  AND dep_time = ?
  AND line_name = ?;
`;
```

iii.
```
//insert new ticket into the ticket table
query = `
INSERT INTO Ticket (ticket_ID, email, flight_num, dep_date, dep_time, line_name)
VALUES (?, ?, ?, ?, ?, ?); `;
connection.query(query, values, (err) => {

    //inserting new purchased value into purchased table
    let purchaseDate = new Date();
    let purchase_date = purchaseDate.toISOString().split("T")[0];
    let purchase_time = purchaseDate.toTimeString().split(" ")[0];
    values = [ticketID, email, card_type, card_num, card_name, card_exp, purchase_date, purchase_time];

    query = `INSERT INTO Purchased (ticket_ID, email, card_type, card_num, card_name, card_exp, purchase_date, purchase_time)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?);`;

    //inserting new purchased value into purchased table
    connection.query(query, values, (err) => {
        return res.json({});
    });
```

iv. On the users side they click on a table of future flights as provided by the search flights user case. Whatever row they click contains the flight's details.

v. Then in this route we join the airplane table with the specific flight the user clicked on to get the number of seats available.

vi. Then we count the number of tickets that exist from the ticket table that match the specific flight.

vii. If there are less tickets than seats, the user can buy a ticket.

viii. So a ticket_ID based on the date is generated and we insert into the ticket table and and the purchased table

d. Cancel Trip

i.
```
let query = `SELECT Ticket.ticket_ID, Flight.status
             FROM Ticket
             JOIN Flight ON
               Ticket.flight_num = Flight.flight_num AND
               Ticket.dep_date = Flight.dep_date AND
               Ticket.dep_time = Flight.dep_time AND
               Ticket.line_name = Flight.line_name
             WHERE Ticket.flight_num = ? AND Ticket.dep_date = ? AND Ticket.dep_time = ? AND Ticket.line_name = ? AND Ticket.email = ?
```

ii.
```
//user can't cancel within 24 hours of the dep_date or after it
let depDate = new Date(`${dep_date}T${dep_time}`);
let now = new Date();
let cutoff = new Date(now.getTime() + (24 * 60 * 60 * 1000));

//if the flight was cancelled, the customer may cancel their ticket regardless of time
if( !(status === "Cancelled" || depDate > cutoff)){
  return res.json({ error: "Cannot cancel within 24 hours of departure or after." });
}

//delete from Purchased first
query = "DELETE FROM Purchased WHERE ticket_ID = ?";
connection.query(query, [ticket_ID], (err) => {

  //delete from Ticket
  query = "DELETE FROM Ticket WHERE ticket_ID = ?";
  connection.query(query, [ticket_ID], (err) => {
    return res.json({ message: "Ticket successfully canceled.", ticket_ID });
```

iii. On the users side they click on a table of flights they have tickets for as provided by the view my flights user case. Whatever row they click contains the flight's details.

iv. We then query for flight status and ticket ID from the ticket table joined with the specific flight the user is referring to. This will find a ticket the user purchased for that flight

v. Then if the flight is not in 24 hours or already passed, the user can cancel the ticket

vi. If the flight was cancelled, the user can also cancel the ticket

vii. So we delete from the purchased and ticket table

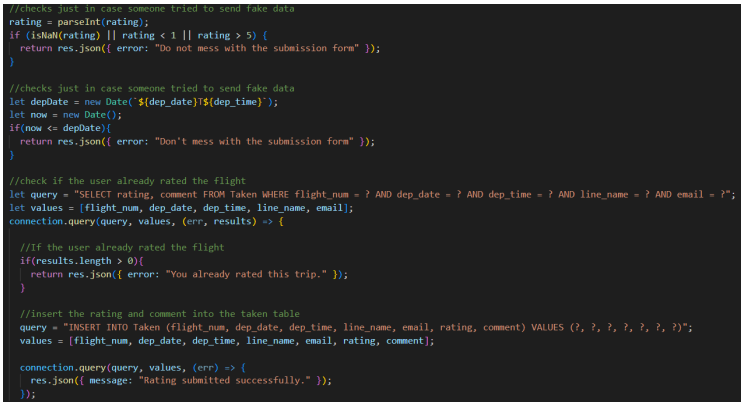
e. Give Ratings and Comment on previous flights

i.
```
let query = `
  SELECT Flight.*
  FROM Ticket
  JOIN Flight ON
    Ticket.flight_num = Flight.flight_num AND
    Ticket.dep_date = Flight.dep_date AND
    Ticket.dep_time = Flight.dep_time AND
    Ticket.line_name = Flight.line_name
  WHERE Ticket.email = ?
`;

connection.query(query, [email], (err, results)=>{
  let now = new Date();

  //check which of these flights the user has taken in the past, not WILL take
  let pastFlights = results.filter((flight) => {
    let depDateTime = new Date(`${flight.dep_date.toISOString().split("T")[0]}T${flight
    return depDateTime <= now;
  });
```

ii.



```
//checks just in case someone tried to send fake data
rating = parseInt(rating);
if (isNaN(rating) || rating < 1 || rating > 5) {
  return res.json({ error: "Do not mess with the submission form" });
}

//checks just in case someone tried to send fake data
let depDate = new Date(`${dep_date}T${dep_time}`);
let now = new Date();
if(now <= depDate){
  return res.json({ error: "Don't mess with the submission form" });
}

//check if the user already rated the flight
let query = "SELECT rating, comment FROM Taken WHERE flight_num = ? AND dep_date = ? AND dep_time = ? AND line_name = ? AND email = ?";
let values = [flight_num, dep_date, dep_time, line_name, email];
connection.query(query, values, (err, results) => {

  //If the user already rated the flight
  if(results.length > 0){
    return res.json({ error: "You already rated this trip." });
  }

  //insert the rating and comment into the taken table
  query = "INSERT INTO Taken (flight_num, dep_date, dep_time, line_name, email, rating, comment) VALUES (?, ?, ?, ?, ?, ?, ?)";
  values = [flight_num, dep_date, dep_time, line_name, email, rating, comment];

  connection.query(query, values, (err) => {
    res.json({ message: "Rating submitted successfully." });
  });
```

iii. First I query all flights the user purchased by joining the flight table with the ticket table that has the users email. Then I filter those results with only the ones who have a dep_date before the current date, so users will only be presented with a table of flights they have taken in the past. You can't rate something you haven't taken yet.

iv. This makes a table on the users end that shows these, and each row contains the details of the flight

v. Then I check if the user tried to send ratings that can't exist or flight information for flights in the future

vi. Then I query to see if the user already rated the current flight by checking the Taken table with flight details that match the one the user clicked on along with the user's email.

vii. Then finally I insert into the taken table

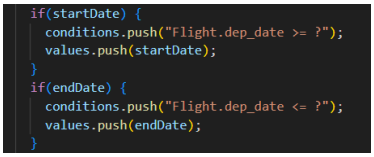
f. Logging out

i. This is done by clicking on a button on the top bar. This sends a request to app.mjs which destroys the session and redirects them back to login

3. Staff Use Cases (staffRouter.mjs

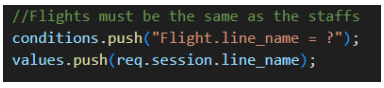
a. View flights

i.



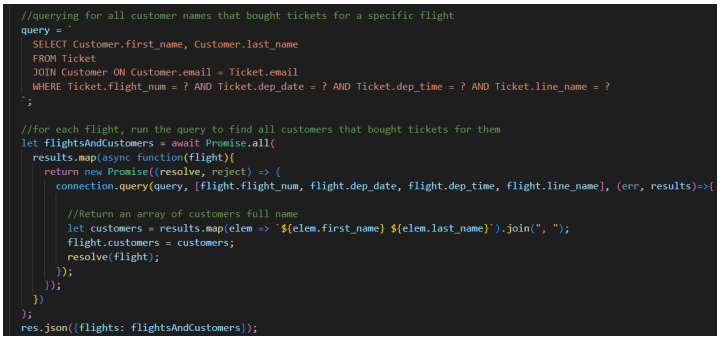
```
  if(startDate) {
    conditions.push("Flight.dep_date >= ?");
    values.push(startDate);
  }
  if(endDate) {
    conditions.push("Flight.dep_date <= ?");
    values.push(endDate);
  }
```

ii.



```
//Flights must be the same as the staffs
conditions.push("Flight.line_name = ?");
values.push(req.session.line_name);
```

iii.



```
//querying for all customer names that bought tickets for a specific flight
query = `
  SELECT Customer.first_name, Customer.last_name
  FROM Ticket
  JOIN Customer ON Customer.email = Ticket.email
  WHERE Ticket.flight_num = ? AND Ticket.dep_date = ? AND Ticket.dep_time = ? AND Ticket.line_name = ?
`;

//for each flight, run the query to find all customers that bought tickets for them
let flightsAndCustomers = await Promise.all(
  results.map(async function(flight){
    return new Promise((resolve, reject) => {
      connection.query(query, [flight.flight_num, flight.dep_date, flight.dep_time, flight.line_name], (err, results)=>{

        //Return an array of customers full name
        let customers = results.map(elem => `${elem.first_name} ${elem.last_name}`).join(", ");
        flight.customers = customers;
        resolve(flight);
      });
    });
  })
);
res.json({flights: flightsAndCustomers});
```

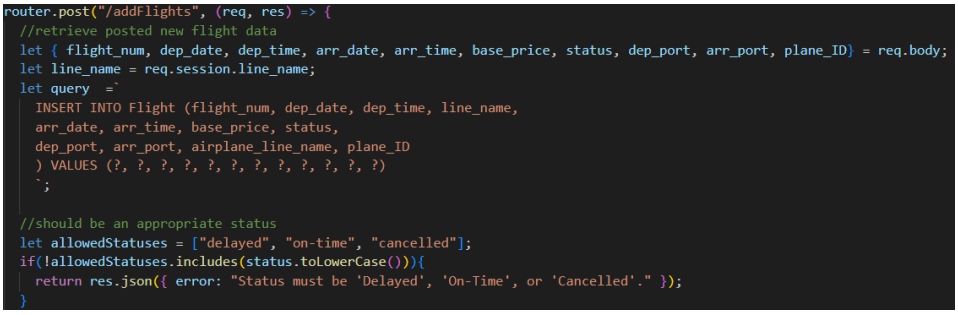
iv. Almost exactly the same as View Public Info except you can search with a start and end date

v. There is also an additional where condition that specifies that all the flights must match the airline the staff member works for

vi. Then, for each flight returned, query for a customer's first and last name from the ticket table joined with the customer table where the tickets flight details match the flight. Add this property to the flight and return this all to the frontend that will display all the flights details with a customers column that shows every customer that bought tickets for each flight

b. Create new flights

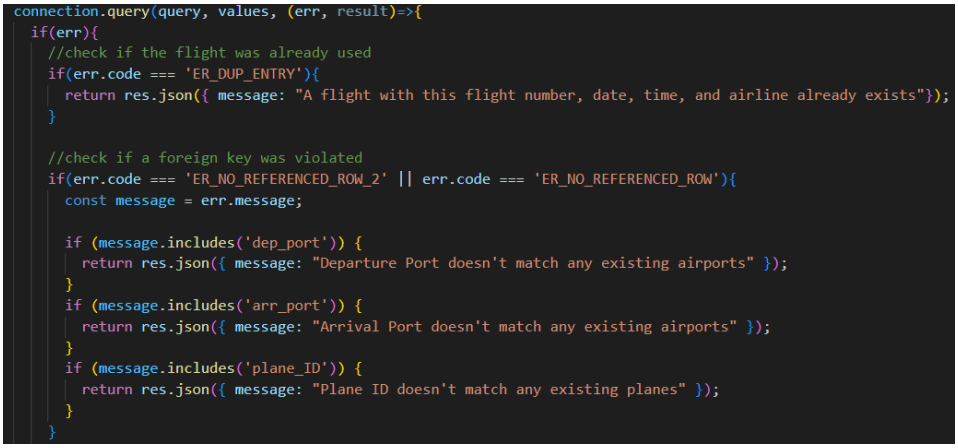
i.



```
router.post("/addFlights", (req, res) => {
  //retrieve posted new flight data
  let { flight_num, dep_date, dep_time, arr_date, arr_time, base_price, status, dep_port, arr_port, plane_ID} = req.body;
  let line_name = req.session.line_name;
  let query  =`
    INSERT INTO Flight (flight_num, dep_date, dep_time, line_name,
    arr_date, arr_time, base_price, status,
    dep_port, arr_port, airplane_line_name, plane_ID
    ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    `;

  //should be an appropriate status
  let allowedStatuses = ["delayed", "on-time", "cancelled"];
  if(!allowedStatuses.includes(status.toLowerCase())){
    return res.json({ error: "Status must be 'Delayed', 'On-Time', or 'Cancelled'." });
  }
```

ii.



```
connection.query(query, values, (err, result)=>{
  if(err){
    //check if the flight was already used
    if(err.code === 'ER_DUP_ENTRY'){
      return res.json({ message: "A flight with this flight number, date, time, and airline already exists"});
    }

    //check if a foreign key was violated
    if(err.code === 'ER_NO_REFERENCED_ROW_2' || err.code === 'ER_NO_REFERENCED_ROW'){
      const message = err.message;

      if (message.includes('dep_port')) {
        return res.json({ message: "Departure Port doesn't match any existing airports" });
      }
      if (message.includes('arr_port')) {
        return res.json({ message: "Arrival Port doesn't match any existing airports" });
      }
      if (message.includes('plane_ID')) {
        return res.json({ message: "Plane ID doesn't match any existing planes" });
      }
    }
```

iii. Insert into the flights table the values passed by the staff. Any errors that occur will be caught and returned the staff, informing them what mistake they made

c. Change Status of flights

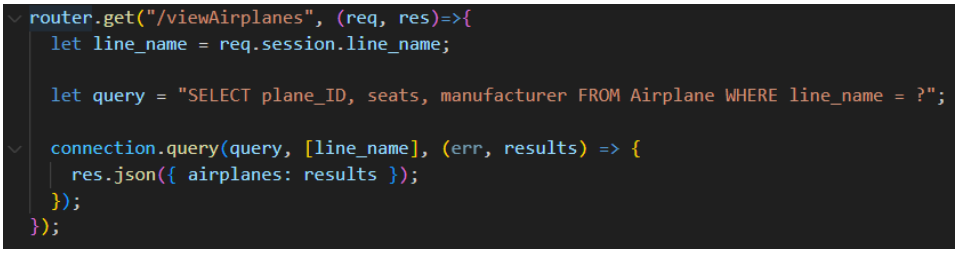
i.

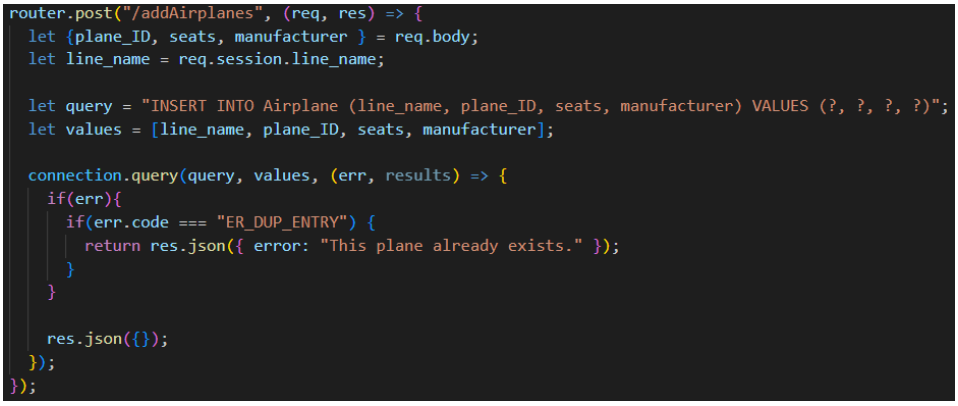

ii. On the staff's side they click on a table of flights of airlines they work for using the view flights use case. When they click on it, specific flight details are passed to this router.

iii. If the posted status is not one of the three defined statuses, the staff is told they made an error

iv. Then we update that flights status and we get the specific flight using the details passed from the staff clicking on a row on the table

d. Add airplane in the system

i.


ii.


iii. Query for all airplanes owned by the staff members' airline.

iv. Then insert into the airplane table a new flight and present an error if that airplane already exists

e. Add new airport in the system

i.

```
//retrieve airports
router.get("/viewAirports", (req, res)=>{
  let query = "SELECT port_code, port_name, city, country FROM Airport";

  connection.query(query, (err, results) => {
    res.json({ airports: results });
  });
})
```

ii.

```
router.post("/addAirports", (req, res) => {
  let {port_code, port_name, city, country} = req.body;
  let query = "INSERT INTO Airport (port_code, port_name, city, country) VALUES (?, ?, ?, ?)";
  let values = [port_code, port_name, city, country];

  connection.query(query, values, (err, results) => {
    if(err){
      if(err.code === "ER_DUP_ENTRY") {
        return res.json({ error: "This port already exists." });
      }
    }
    res.json({});
  });
});
```

iii. First query for all airports (just for visual feedback when you add a new airport)

iv. Then insert into the airport table a new airport and present an error if that airplane already exists

f. View flight ratings

i.

```
router.post("/viewRatings", (req, res) => {
  let line_name = req.session.line_name;

  const query = `
    SELECT
      F.flight_num,
      F.dep_date,
      F.dep_time,
      F.line_name,
      ROUND(AVG(T.rating), 2) AS average_rating
    FROM Flight F
    LEFT JOIN Taken T
      ON F.flight_num = T.flight_num
      AND F.dep_date = T.dep_date
      AND F.dep_time = T.dep_time
      AND F.line_name = T.line_name
    WHERE F.line_name = ?
    GROUP BY F.flight_num, F.dep_date, F.dep_time, F.line_name
    ORDER BY F.dep_date ASC, F.dep_time ASC;
  `;

  connection.query(query, [line_name], (err, results) => {
    res.json({ flights: results });
  });
});
```

ii.

```
let query = `
  SELECT Customer.first_name, Customer.last_name, Taken.rating, Taken.comment
  FROM Taken
  JOIN Customer ON Taken.email = Customer.email
  WHERE Taken.flight_num = ? AND Taken.dep_date = ? AND Taken.dep_time = ? AND Taken.line_name = ?
`;
```

iii. Query for the average rating from the flight table joined with the taken table grouped by the flight and ordered by date

iv. This will make a table that shows minimal flight details along with the average rating

v. Then when you click on a row in that table, a new query is made that retrieves all customers that have taken that specific flight, since the row passes on the details of that flight.

vi. You use these flight details as where conditions and then join the customer table with the taken table to find the names of the customers who took that flight

g. View reports

i.
```
let query = `
SELECT
  YEAR(purchase_date) AS year,
  MONTH(purchase_date) AS month_number,
  MONTHNAME(purchase_date) AS month_name,
  COUNT(*) AS tickets_sold
FROM Purchased
JOIN Ticket ON Purchased.ticket_ID = Ticket.ticket_ID
WHERE Ticket.line_name = ?`;
```

ii.
```
let values = [line_name];
if(startDate){
  query += " AND purchase_date >= ?";
  values.push(startDate);
}
if(endDate){
  query += " AND purchase_date <= ?";
  values.push(endDate);
}
```

iii.
```
//group by the year and month
query += " GROUP BY year, month_number, month_name ORDER BY year ASC, month_number ASC";
```

iv. Get the year, month, and count of all tickets sold by joining the purchased table (to get purchase date and tickets table) with the ticket table to get the line it belongs to so we only see tickets purchased by the same airline as the staff member.

v. We also specify a range of dates and then group by the year and month and order by year and month.

h. Logging out
   i. This is done by clicking on a button on the top bar. This sends a request to app.mjs which destroys the session and redirects them back to login

4. Additional Requirements

a. You should implement Air ticket reservation system as a web-based application
   i. Done using node, express, and mysql server
b. Enforcing complex constraints:
   i. Done using isAuthenticated in my authenticationRoutes.mjs
   ii. Users that are not logged in are redirected back to login
   iii. Customers cannot access staff pages and are redirect back to their home page
   iv. Customers and staff cannot mess with other customers and staff because the backend always uses session data to determine who they are.
c. Session Management:
   i. Implemented and all customer and staff information is stored in sessions
   ii. isAuthenticated always checks if the user allowed to access certain routes and all routes verify which customer or staff is trying to perform an action using their session email/username
d. You should take measures to prevent cross-site scripting vulnerabilities

    i. Handlebars automatically escapes text passed in, so users cannot pass script in
e. The user interface should be usable, but it does not need to be fancy. For each type of users, you need to implement different home pages where you only show relevant use cases for that type of users and you should not show/combine all the use cases in one page
    i. All use cases have their own pages