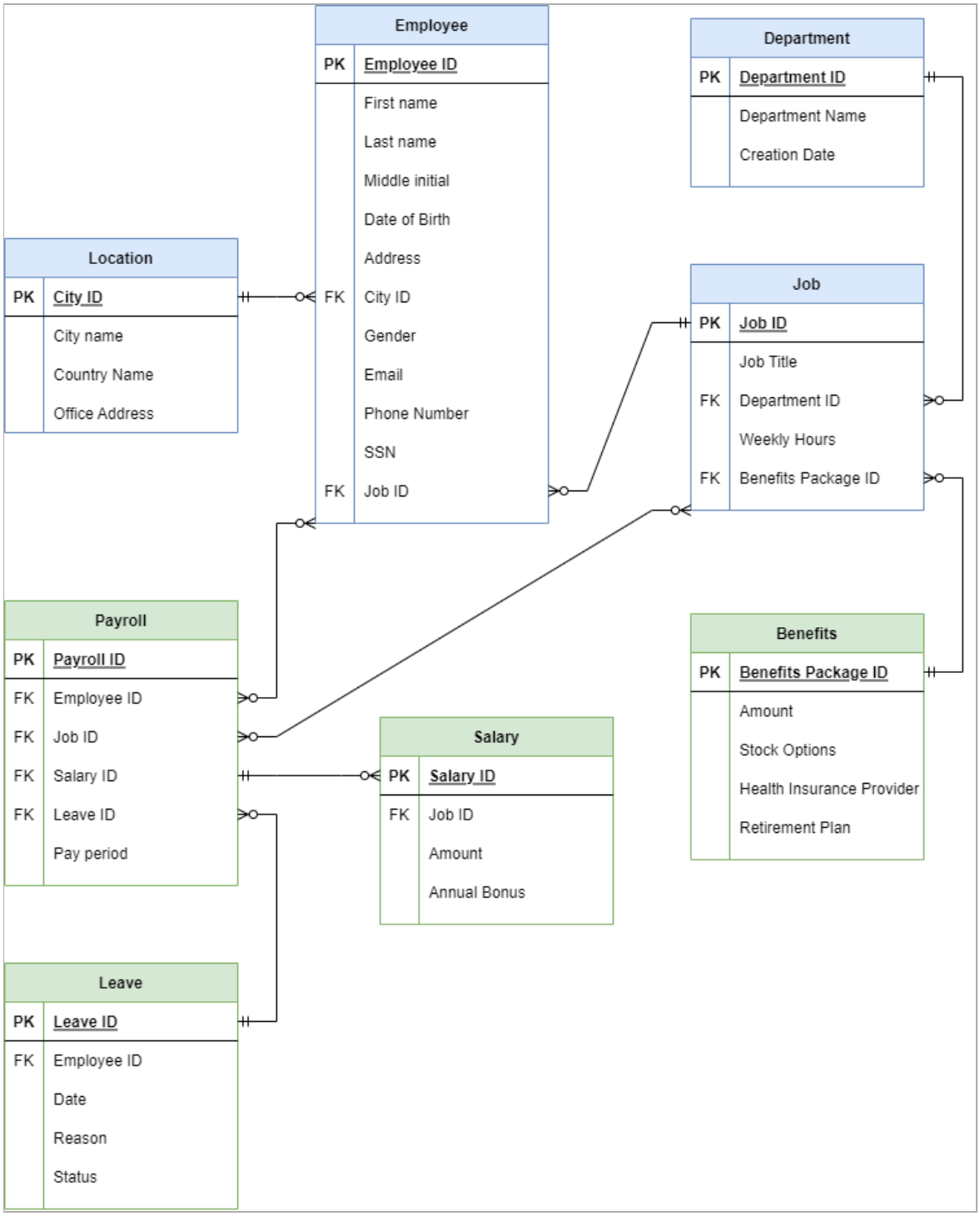
Syed Rizvi
Nathan Valtierra
Kurmanbek Bazarov

CS 3380
Dr. Carlos Ordonez

Team 07 HW3 Final Report

A. ER Model Diagram

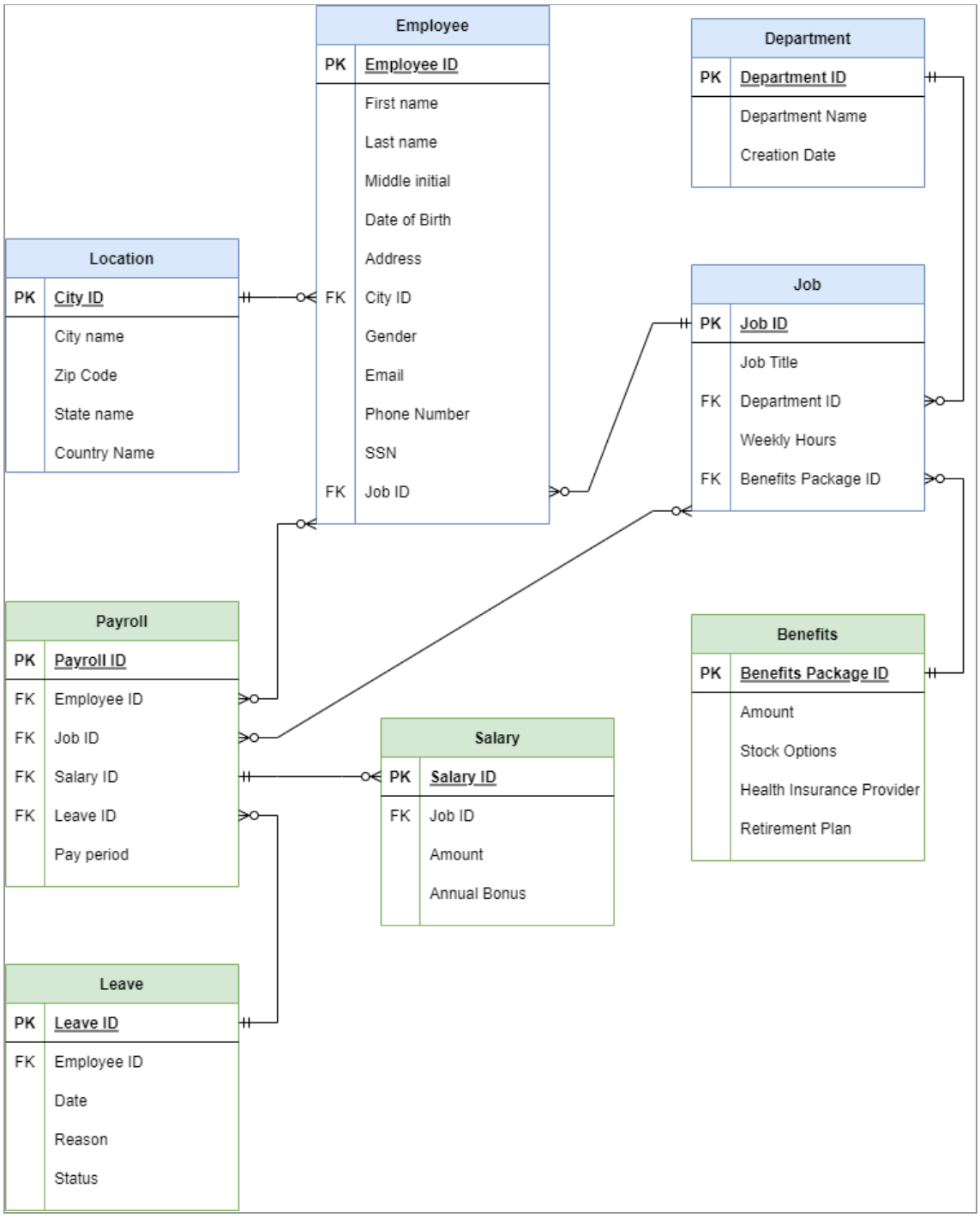
Initial ER Diagram:



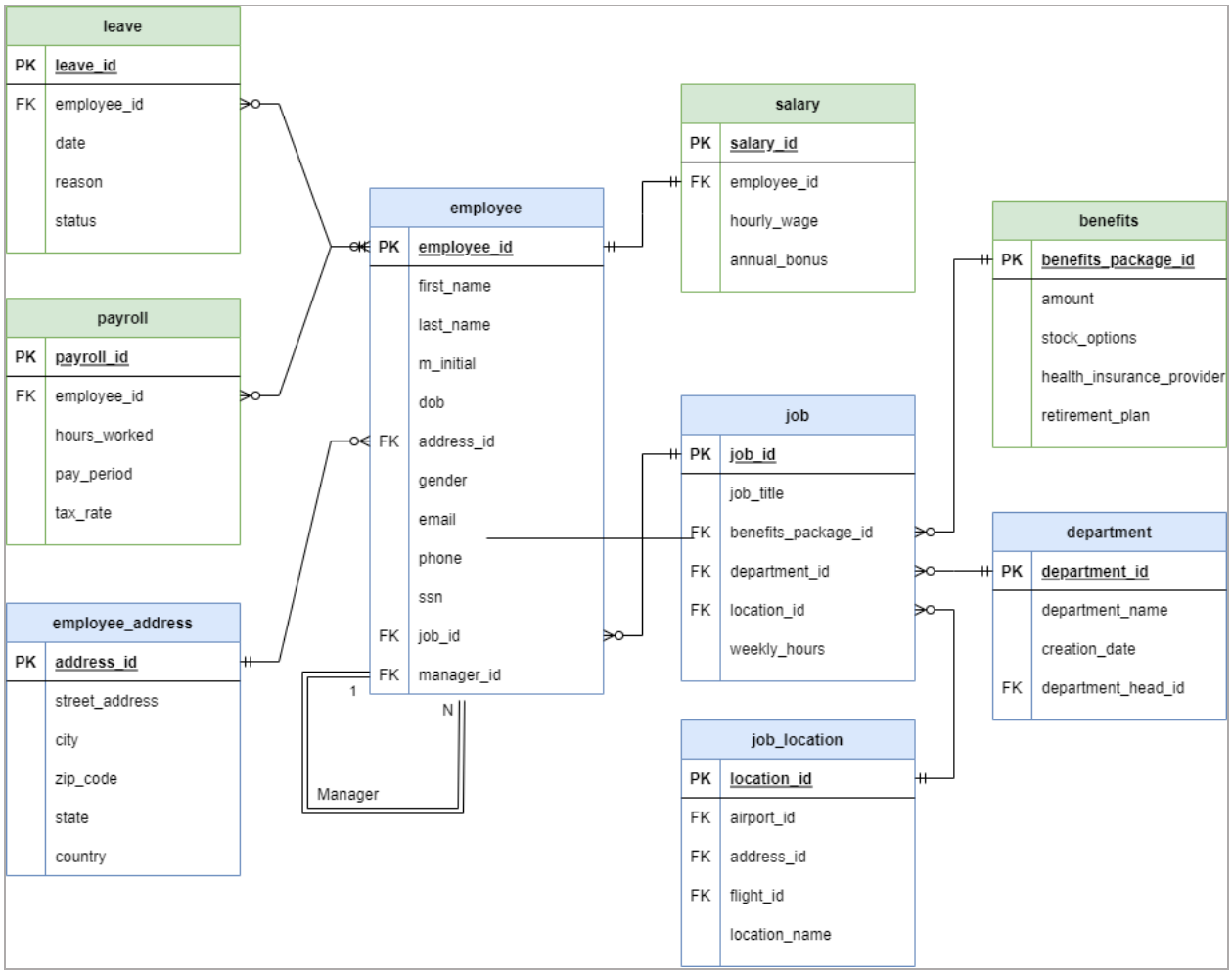
Syed Rizvi
Nathan Valtierra
Kurmanbek Bazarov

CS 3380
Dr. Carlos Ordonez

Intermediate ER Diagram:



Final ER Diagram:



B. Discussion on Final ER Diagram Normalization, Attribute Redundancy, Real-world Use, and Entity Relationships

As we were designing our ER model to fit the considerations and use cases of enterprise airline application, we had to make decisions as to whether or not to normalize each relation fully up to 3rd Normal Form, or leave a few functional dependencies in some relations for the sake of storage and access considerations. As a result, some of the relations in our ER model (namely employee_address) were left with some functional dependencies disqualifying it for third normal form, because keeping the attributes provided better access to attributes that were often accessed together. For our employee_address relation, we could have further normalized the table by decomposing things like state and country to separate tables with their own IDs; this would have gotten rid of the state -> country functional dependency, however we determined that the state and country name would often be accessed together when retrieving

employee address information, and since the scope of this project is only for design purposes, we decided not to create small lookup tables for state and country.

Regarding attribute redundancy, the main redundancy within our ER model diagram is the use of Employee ID as a foreign key in multiple relations. This was done in the context of how we decided to calculate payroll by joining together information about an employees job, work hours, and hourly salary. With our current design, there are no orphaned tables in our ER model and we are able to calculate payroll for employees through querying and joining without having to store many additional foreign keys in payroll. We recognize, however, that our payroll calculations were done on a small scale in our project, and a real enterprise database application would likely need many more foreign keys and joins to bring together all the information to run payroll for an entire company. As mentioned earlier, we also recognize that for real-world use our ER model would need to transition to having many more lookup tables (e.g. city, state, airport_id linking to an airport relation, address IDs) that we did not include in our project due to the limited scope of our target application as well as the limited amount of time and manpower we had to develop this project.

Regarding relationships between entities in our ER model, we mostly used one to (optional) many relationships between relations such as employee and leave, or employee and job. Our reasoning for this was that one employee may have many leave entries in the database, each leave entry will only have one employee attached to it, and an employee does not have to have a leave entry in order to be in the database, thus justifying a one to (optional) many relationship. For cases such as employee and job, our assumption is that one employee can have one job, but each job entry can have many employees assigned that job (i.e. there could be many junior software developers at Oracle at their Austin office). We recognize that in the real-world companies may have more complex or different ways of relating employees to jobs, which might change the relationship, however for the sake of simplicity for this project we operated under this assumption.

C. Web App Summary

The web interface of our application was developed using the React JS library, which integrates JavaScript code, HTML, and CSS in order to make a responsive, well-structured frontend application. Our choice to use React rather than plain HTML with JavaScript reflects the skillset of our team members (we had one member in our group who was an experienced React web developer) as well as the useful setup of React class components that easily allows us to organize our web interface code into components, classes, and functions that make requests to the backend Node JS/Express server.

The web interface consists of nested routes, each of which renders interface components to the end user. For example, the URL extension ‘/’ will render the top navigation bar, ‘/employees’ renders the top navigation bar in addition to the left side panel for choosing Employee forms, and finally ‘/employees/search’ will render the top navigation bar, the left side panel for employees, and the search employees form. Note that the nested React routing sequence is separate from Express backend server API endpoints (for example, the frontend search form at ‘/employees/search’ may make a request to the backend server endpoint ‘/employee/:id’). This works because React routing is separate from Express backend routing in the web application.

Upon starting the frontend application, the user will be greeted with a minimal screen with the top navigation bar, where they can begin navigating to web forms for common HR operations such as retrieving employee details, assigning a job to a department, and running payroll processes. Whenever a form is submitted from the web interface, a request is made to the backend server which processes the inputted information, generated the appropriate SQL and transactions, sends the queries to the database, and logs any queries and transactions in log files. Nearly all core processing is done by the database itself (e.g. calculating payroll amounts is done through database querying), and the output of the queries is returned to the web interface to be shown back to the user. Error handling and input validation on all forms is done through HTML and JavaScript checkout and error handling logic, and there is an option on every web form once submitted to show the SQL queries that were used in the previous submission.

D. Web App Considerations, Discussion, and Real-World Scaling

As we were developing the web interface of our application, we kept in mind the consideration that an HR employee at an airline would be using our web interface to interact with the airline database and complete regular operations such as adding employees, running payroll, and updating location information. We thus focused on clean, minimal web forms with input validation and error popups in our web design, along with efficient communication with the server backend through GET, POST, and PUT requests. We maintain good design, for example by ensuring that information processing is done in the server backend/database while the frontend interface is simply for taking user input, validating it, and displaying the returned information from the server querying and processing. We included a button on each form to show the SQL queries that were run for each form submission, however this was done for the purposes of grading and checking, and in a real-world setting we would ensure that the user never interacts directly with database and server backend outputs.

Regarding scaling, we took care to formulate database requests properly as transactions or simple queries based on whether concurrent user access would be a consideration for the operation. For example, updating an employee's information fields is done through a transaction, so that when multiple updates are made concurrently Postgres can handle the concurrent access and maintain consistent database states. Our application is thus designed in mind with scaling to multiple users, which is a needed feature for a system on the scale of an airline application. If we were to scale it, however, we would take more measures and time in designing the frontend in order to accommodate every person that may use a website in the real world. For the purposes of this project, our web application was designed for effective communication with the server backend and information processing through requests to backend server endpoints, however to scale it to the real-world we would need to start considering much harder considerations such as load-balancing, advanced request handling, and so on.

E. Important Transactions and Queries

In this section, we will provide a list of actions within our web-database application that uses transactions to handle processing. We will also provide a few examples of important transaction code and import SQL queries. More transactions and query information can be found in 'important_queries.sql' file in our code, as well as in the query.sql and transaction.sql files within the 'db' folder of the code base as you are using the application and making requests.

Transaction Examples

```
/* Create a new payroll entry for employee 1000000 who worked for 360 hours in the month of
November 1979 with a 10% tax rate*/
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

INSERT INTO payroll (employee_id, hours_worked, pay_period, tax_rate, gross_income, taxed_income,
net_income)
SELECT  employee_id,
      hours_worked,
      pay_period,
      tax_rate,
      gross_income,
      gross_income * tax_rate       AS taxed_income,
      gross_income * (1 - tax_rate) AS net_income
FROM (
        SELECT  e.employee_id,
                '360'::INT         AS hours_worked,
                 '1979-11-01'::DATE  AS pay_period,
                 '0.1'::REAL        AS tax_rate,
                    CASE
                        WHEN '360'::INT <= 4.4 * weekly_hours THEN '360'::INT * hourly_wage
                        ELSE (1.5 * '360'::INT - 0.5 * 4.4 * weekly_hours) * hourly_wage
                    END gross_income
        FROM employee e
                JOIN job j    ON e.job_id = j.job_id
                JOIN salary s ON e.employee_id = s.employee_id
        WHERE NOT e.job_id = 0
          AND e.employee_id = '1000000'
) AS gross_calc;

COMMIT;
END TRANSACTION;

-- Notes about transaction: No overtime, and 50% more pay for overtime hours


/* Inserting a new employee into Database */
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO employee_address (street_address,city,country,zip_code,state)
        VALUES ('12345 SEVENTH STREET','HOUSTON','UNITED STATES','28822','TEXAS')
        RETURNING address_id;
INSERT INTO employee
        (first_name,last_name,dob,gender,address_id,ssn,phone,email,job_id,manager_id)
        VALUES
```

```
        ('SYED','RIZVI','2000-01-01','M','268','123456677','+18328323322','syed.a.rizvi@email.com','9','
        1000000')
        RETURNING employee_id;
INSERT INTO salary (employee_id,hourly_wage,annual_bonus)
        VALUES ('1000264','18.00','2000');
COMMIT;
END TRANSACTION;
```

Query Examples:

```
-- Example: Getting a summary of the first 10 managers who work in the finance department
SELECT  COUNT(*),
      m.employee_id,
      m.first_name,
      m.m_initial,
      m.last_name,
      job_title,
      department_name
FROM employee e
        JOIN employee m   ON e.manager_id = m.employee_id
        JOIN job j        ON m.job_id = j.job_id
        JOIN department d ON j.department_id = d.department_id
WHERE department_name LIKE 'FINANCE%'
GROUP BY  m.employee_id,
       m.first_name,
       m.m_initial,
       m.last_name,
       job_title,
       department_name
ORDER BY job_title ASC
OFFSET 0
LIMIT 10;
```

```
/* Getting department information, including the number of jobs and employees in the department */
SELECT  COUNT(DISTINCT j.job_id) AS job_count,
      COUNT(DISTINCT e.employee_id) AS employee_count,
      d.department_id,
      department_name,
      creation_date,
      department_head_id,
      h.first_name,
      h.m_initial,
      h.last_name
FROM department d
        JOIN job j      ON j.department_id = d.department_id
```

```
        JOIN employee e ON e.job_id = j.job_id
        JOIN employee h ON h.employee_id = d.department_head_id
GROUP BY  d.department_id,
    department_name,
    creation_date,
    department_head_id,
    h.first_name,
    h.m_initial,
    h.last_name
ORDER BY department_id ASC
OFFSET 0
LIMIT 10;
```

Important operations in web app that use transactions:

- Creating a new benefits package, department, employee, job, job location, leave entry, or payroll entry
- Changing or updating the details of a benefits package, department, employee, job, leave entry, or payroll entry
- Deleting a department, employee, or job
- Running payroll for all existing employees
- Reinitializing the entire database through the menu option

F. Division of Responsibilities Within Team 07

Given the skillsets of our members, we decided on the following distribution of tasks in order to complete as many features of the airline HR system as possible given the project timeframe:

- All members participated in discussions regarding the iterative development of the ER model
- Syed Rizvi (dbs098) developed the React frontend interface, including all web forms, styling, and requests made to the backend server
- Nathaniel Valtierra (dbs115) developed the Express and Node JS backend server, including all requests made to the database
- Kurmanbek Bazarov helped design transaction ideas with Nathaniel, interfaced with the professor throughout the course of the project, and tested the React frontend for web form errors

G. Additional Information for Setting up and Running Code

Detailed steps on how to download, set up, and run our web application and server backend code are in the ReadME file of out project GitHub page. The link is posted below, along with basic pseudo steps explained in this pdf:

Project Homepage & ReadME:
https://github.com/SyedA5688/3380_airline_DB_system#3380-airline-hr-database-system-project

Demo video:
https://youtu.be/K2cJQ1I8Hcc

Basic setup steps:

1. Download the code from the GitHub repository (or from the hw3 folder of team member Kurmanbek Bazarov, dbs006 on the 3380db.cs.uh.edu linux server)
2. Install dependencies: Run ‘npm install’ in the base directory of our application code
3. If not already present, create a file called ‘.env’ in the base directory and enter the following lines of connection information into it. This will connect to the PostGres database of our team member Kurmanbek Bararov (dbs006)
    a. PGHOST='3380db.cs.uh.edu'
    b. PGUSER='dbs006'
    c. PGDATABASE='COSC3380'
    d. PGPASSWORD=’dbs006’
    e. PGPORT=5432
4. Start the application: Run ‘node index.js’ in the base directory

The database used for our application was originally an online Postgres database hosted on ElephantSQL. We afterwards migrated all our tables and data to the 3380db.cs.uh.edu server, where it now resides. The connection information to connect our application to this database is already present in our code, however if anyone would like to recreate our database somewhere else we will provide an SQL script to recreate our entire database with prepopulated data rows in our code (name will be create_table_script.sql).