CS2102 PROJECT TEAM 56

Table Of Content


```
LEOW JIT YONG (A0182998A)
LIM WEI DONG (A0184149X)
HOON CHEE PING (A0183922B)
LEE EE JIAN (A0183105N)
```


1. Roles and Responsibilities

Leow Jit Yong - Fullstack Developer

Lim Wei Dong - Fullstack Developer

Hoon Chee Ping - Fullstack Developer

Lee Ee Jian - Fullstack Developer

2. Application

2.1. Data Requirements and Constraints

- Each user entity stores the userId, the user's name, and date of creation of account
- Each user is uniquely identifiable by userId (primary key)
- Each user must be one of : (i) Customer (ii) Restaurant Staff (iii) Rider (iv) FDS Manager
- Each customer/rider/restaurant staff/FDS manager is identified by userId (primary key) which is referenced from users (foreign key)
- Each customer entity is a user, stores userId and credit card information
- Each rider entity is a user, stores userId and the area he/she lives in
- Each rider must be either : (i) Full-Time Rider (ii) Part-Time Rider
- Each part-time rider works on a weekly work schedule (WWS)
- Each full-time rider works on a monthly work schedule (MWS)
- Each restaurant staff entity is a user, and stores userId and restaurant name he/she works for
- Every restaurant staff must work in exactly one restaurant
- Each FDS manager entity is a user and stores the userId
- Each restaurant entity stores the restaurant name, area of locality, and a minimum order amount of any order
- Each restaurant is uniquely identifiable by its restaurant name (primary key)
- Each food entity stores the food name and category of the food
- Each food is uniquely identified by its food name (primary key).
- Each food entity sold by a restaurant entity is captured in the Sells relation
- Each sells relation stores the restaurant name, food name, the price, and quantity available for each food
- Each sells relation is uniquely identified by the restaurant name coupled with the food name (primary key)
- Each sells relation references restaurant name from restaurants (foreign key) and food name from the food (foreign key)
- Each restaurant can create promotions for their menu items
- Each promotion entity stores the promo code, the description, the creator of the promotion, the restaurant name it is applicable to, the unit of measurement of the discount, the rate of discount, and the start and end date of the promotion
- Each promotion is uniquely identifiable by the promo code and the name of the restaurant it is applicable to (primary key).
- Each order entity stores the orderId, userId of the customer, the promotional code used, the restaurant name that the promo code is applicable to, the mode of payment by the customer, time of order being placed, delivery location, and reward points being used to offset the price
- Each order is uniquely identified by the orderId (primary key)
- Each order references the userId of the customer who created the order (foreign key)
- Each order references the promo code, together with the restaurant name that the promo code is applicable to (foreign key) from promotions entity

- Each order must be delivered exactly once by a rider
- Each delivers relation stores the orderId for the order being delivered, the userId of the rider, the time rider departs for the restaurant, the time rider arrives at the restaurant, the time rider leaves the restaurant, the delivery time to the customer, and the rating received for the delivery
- Each deliver is uniquely identified by orderId (primary key) referenced from orders (foreign key), and references rider for userId (foreign key)
- Each order contains food items from a single restaurant
- Each contains aggregate relation stores the orderId it belongs to, the restaurant name and food name of the food, the quantity of the food ordered, and the review for the specific food
- Each contains entry is uniquely identified by orderId,the restaurant name and food name (primary key)
- The restaurant and food name is referenced by the sells relation (foreign key), and the orderId is referenced from the orders entity (foreign key)
- Each Weekly Work Schedule (WWS) stores the scheduleId, userId of the rider, start date and end date of the schedule.
- Each WWS is uniquely identifiable by it's scheduleId (primary key), and belongs to a specific rider which is referenced by userId (foreign key)
- Each Monthly Work Schedule stores the scheduleId of 4 unique WWS scheduleId
- Each MWS is uniquely identifiable by the 4 scheduleId of the WWS it consists of (primary key), which is referenced from the WWS(foreign key)
- Each WWS is made up of work intervals
- Each interval entity stores the intervalId, scheduleId of the WWS it belongs to, start time and end time of interval
- Each interval is uniquely identifiable by intervalId (primary key), and must belong to exactly one WWS that is referenced by scheduleId (foreign key)

2.2. Functionalities

Customer

Each customer should be able to create and update their user account details. When making an order, customers are able to browse for food items by (i) area (ii) restaurant (iii) food name (iv) food category. Upon making their order, customers can apply promotion codes and rewards points to their order. Finally, each customer should be able to view their past order history, and reviews of food items from restaurant menu.

Restaurant Staff

Each restaurant staff should be able to create and update their user account details. Restaurant staff should be able to view and update their restaurant menu. Restaurant staff should be able to create new promotions for their restaurant, and view summary information of previous promotions. This includes the duration of the promotion, the total cost of order received, and the total number of orders received during promotion period. Finally, they should also be able to view a monthly summary for order information. This includes the total number of orders, the cost of the orders, as well as the top 5 food choices for the month.

Riders

Each rider should be able to create and update their account details. Riders should be able to view their delivery history, as well as their work schedule history. They should also be able to declare weekly work schedules (WWS) (for part-timers) or their monthly work schedules (MWS) (for full-timers). The schedules declared should be in line with the FDS policy requirements (e.g. between 10 and 48 hours each week). Finally, they should be able to view summary information for a particular month. This includes total salary, delivery fees earned, hours worked, average rating, average delivery time, and number of deliveries for the month.

FDS Manager

Each FDS manager should be able to create and update their account details. FDS managers should be able to register restaurants into the application. FDS managers should also be able to view monthly summary information. This includes total number of new customers, total number of orders made, and the total cost of all orders. FDS managers should also be able to view an hourly summary information. This includes the total number of orders for each location, and the number of riders for each interval. Finally, FDS managers should be able to view the monthly summary information that individual customers and riders have access to.

3. ER Model

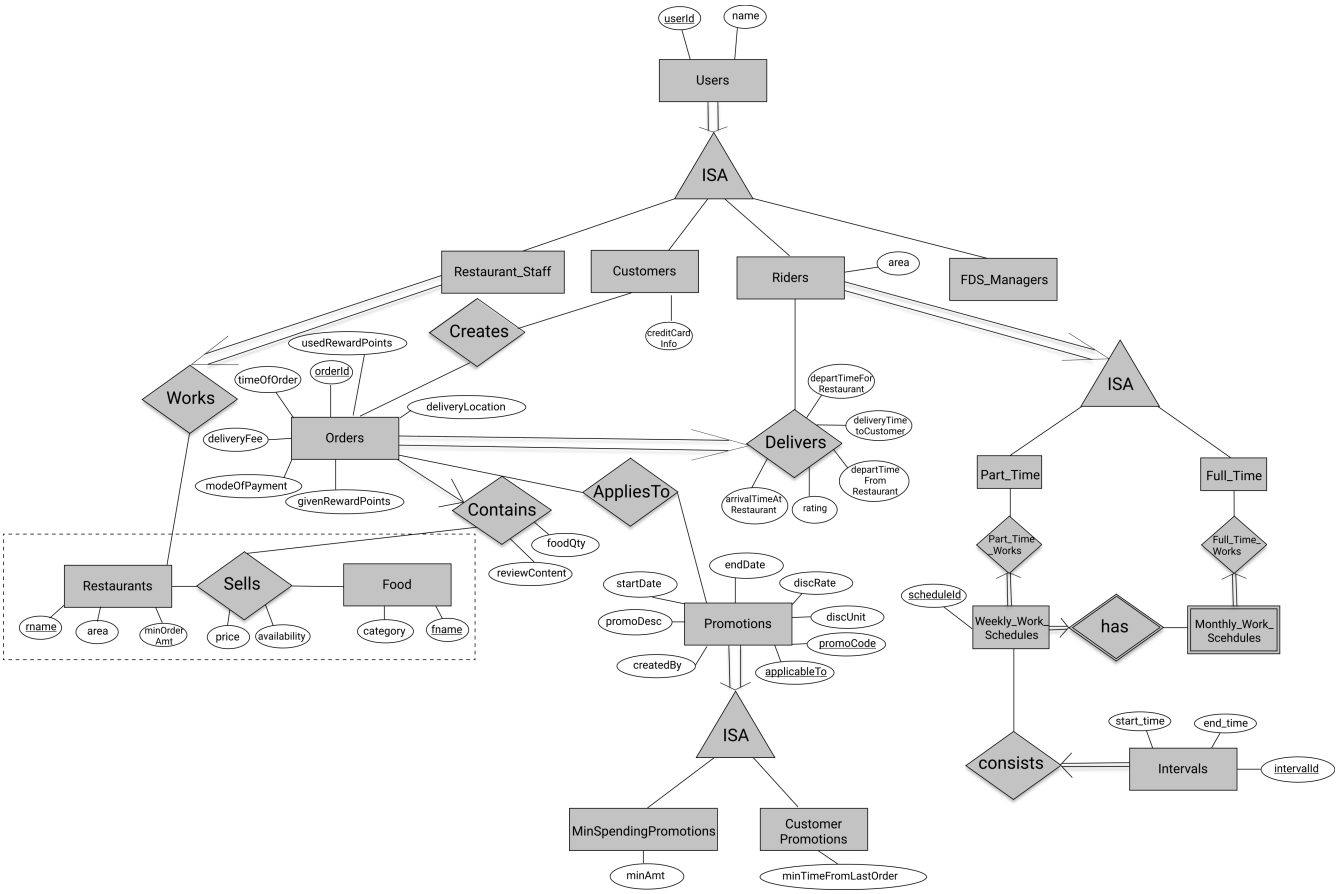

Figure 1. ER diagram for the Food Delivery Service application

3.1. Design considerations

Promotions - Is an ISA relation to all promotion types. By abstracting out key attributes that are common to all promotions, we are able to achieve extensibility to easily create more types of promotions.

Monthly Work Schedules (MWS) - By implementing monthly work schedules (MWS) such that it is composed of 4 unique weekly work schedules (WWS), we can leverage on triggers and checks that are done for the WWS which also apply to each week of the MWS. Checking that each week of the MWS is equivalent is also efficient because we can simply replicate one WWS for 4 times with only the dates adjusted for.

4. Database Relational Schema

Users: BCNF

```
CREATE TABLE Users (
    userId      SERIAL,
    name        VARCHAR(100),
    dateCreated TIMESTAMP,
    PRIMARY KEY (userId)
);

Non-trivial FDs F = {userId → name}
```

Restaurants: BCNF

```
CREATE TABLE Restaurants (
    rname           VARCHAR(200),
    minOrderAmt     NUMERIC(8, 2),
    area            VARCHAR(20),
    PRIMARY KEY (rname),
    CHECK(area = 'central' OR
        area = 'west' OR
        area = 'east' OR
        area = 'north' OR
        area = 'south')
);

Non-trivial FDs F = {rname → (minOrderAmt)(area)}
```

Food schema : BCNF

```
CREATE TABLE Food (
    fname           VARCHAR(20),
    category        VARCHAR(20) NOT NULL,
    PRIMARY KEY (fname),
    CHECK (category = 'western' OR
        category = 'chinese' OR
        category = 'japanese' OR
        category = 'korean' OR
        category = 'fusion')
);

Non-trivial FDs F = {fname → category}
```

Sells schema : BCNF

```
CREATE TABLE Sells (
    rname           VARCHAR(20) REFERENCES Restaurants
                            on DELETE CASCADE
                            on UPDATE CASCADE,
    fname           VARCHAR(20) REFERENCES Food
                            on DELETE CASCADE
                            on UPDATE CASCADE,
    price           NUMERIC(8, 2) NOT NULL,
    availability    INTEGER DEFAULT 10,
    PRIMARY KEY (rname, fname)
);

Non-trivial FDs F = {(fname)(rname) → (price)(availability)}
```

FDS Manager schema : BCNF

```
CREATE TABLE FDS_Managers (
    userId      INTEGER,
    PRIMARY KEY (userId),
    FOREIGN KEY (userId) REFERENCES Users
                on DELETE CASCADE
                on UPDATE CASCADE
);

Non-trivial FDs F = {0}
```

Restaurant Staff schema : BCNF

```
CREATE TABLE Restaurant_Staff (
    userId      INTEGER,
    rname       VARCHAR(20) REFERENCES Restaurants
                            on DELETE CASCADE
                            on UPDATE CASCADE,
    PRIMARY KEY (userId),
    FOREIGN KEY (userId) REFERENCES Users
                            on DELETE CASCADE
                            on UPDATE CASCADE
);

Non-trivial FDs F = {userId → rname}
```

Customers schema : BCNF

```
CREATE TABLE Customers (
    userId              INTEGER,
    creditCardInfo      VARCHAR(100),
    PRIMARY KEY (userId),
    FOREIGN KEY (userId) REFERENCES Users
                            on DELETE CASCADE
                            on UPDATE CASCADE
);

Non-trivial FDs F = {userId → creditCardInfo}
```

Riders schema : BCNF

```
CREATE TABLE Riders (
    userId      INTEGER,
    area        VARCHAR(20) NOT NULL,
    PRIMARY KEY (userId),
    FOREIGN KEY (userId) REFERENCES Users
                            on DELETE CASCADE
                            on UPDATE CASCADE,
    CHECK(area = 'central' OR
        area = 'west' OR
        area = 'east' OR
        area = 'north' OR
        area = 'south')
);

Non-trivial FDs F = {userId → area}
```

Part-time schema : BCNF

```
CREATE TABLE Part_Time
(
    userId              INTEGER,
    PRIMARY KEY (userId),
    FOREIGN KEY (userId) REFERENCES Riders
                            on DELETE CASCADE
                            on UPDATE CASCADE
);

Non-trivial FDs F = {0}
```

Full-time schema : BCNF

```
CREATE TABLE Full_Time
(
    userId               INTEGER,
    PRIMARY KEY (userId),
    FOREIGN KEY (userId) REFERENCES Riders
                            on DELETE CASCADE
                            on UPDATE CASCADE
);

Non-trivial FDs F = {0}
```

Weekly Work Schedules (WWS) schema : BCNF

```
CREATE TABLE Weekly_Work_Schedules
(
    scheduleId              SERIAL,
    userId                  INTEGER,
    startDate               TIMESTAMP,
    endDate                 TIMESTAMP,
    PRIMARY KEY (scheduleId),
    FOREIGN KEY (userId) REFERENCES Riders (userId),
        check ((endDate::date - startDate::date) = 6)
);

Non-trivial FDs F = {scheduleId → (userId)(startDate)(endDate)}
```

Monthly Work Schedules (MWS) schema : BCNF

```
CREATE TABLE Monthly_Work_Schedules (
    scheduleId1             INTEGER REFERENCES Weekly_Work_Schedules
                                    ON DELETE CASCADE,
    scheduleId2             INTEGER REFERENCES Weekly_Work_Schedules
                                    ON DELETE CASCADE,
    scheduleId3             INTEGER REFERENCES Weekly_Work_Schedules
                                    ON DELETE CASCADE,
    scheduleId4             INTEGER REFERENCES Weekly_Work_Schedules
                                    ON DELETE CASCADE,
    PRIMARY KEY (scheduleId1, scheduleId2, scheduleId3, scheduleId4)
);

Non-trivial FDs F = {scheduleId1 → (scheduleId2)(scheduleId3)(scheduleId4)
                     scheduleId2 → (scheduleId1)(scheduleId3)(scheduleId4)
                     scheduleId3 → (scheduleId1)(scheduleId2)(scheduleId4)
                     scheduleId4 → (scheduleId1)(scheduleId2)(scheduleId3)}
```

Intervals schema : BCNF

```
CREATE TABLE Intervals
(
    intervalId              SERIAL,
    scheduleId              INTEGER,
    startTime               TIMESTAMP,
    endTime                 TIMESTAMP,
    PRIMARY KEY (intervalId),
    FOREIGN KEY (scheduleId) REFERENCES Weekly_Work_Schedules (scheduleId)
                                ON DELETE CASCADE,
        check (DATE_PART('minutes', startTime) = 0
        AND DATE_PART('seconds', startTime) = 0
        AND DATE_PART('minutes', endTime) = 0
        AND DATE_PART('seconds', startTime) = 0
        AND DATE_PART('hours', endTime) - DATE_PART('hours', startTime) <= 4
        AND startTime::date = endTime::date
        AND DATE_PART('hours', endTime) > DATE_PART('hours', startTime)
        AND startTime::time >= '10:00'
        AND endTime::time <= '22:00'
        )
);

Non-trivial FDs F = {intervalId → (scheduleId)(startTime)(endTime)}
```

Promotions schema : BCNF

```
CREATE TABLE Promotions (
    promoCode       VARCHAR(20),
    promoDesc       VARCHAR(200),
    createdBy       VARCHAR(50),
    applicableTo    VARCHAR(200) REFERENCES Restaurants(rname)
                                    ON DELETE CASCADE,
    discUnit        VARCHAR(20) NOT NULL,
    discRate        VARCHAR(20) NOT NULL,
    startDate       TIMESTAMP NOT NULL,
    endDate         TIMESTAMP NOT NULL,
    PRIMARY KEY (promoCode, applicableTo),
    CHECK (discUnit = '$' OR discUnit = '%' OR discUnit = 'FD')
);


Non-trivial FDs F = {(promoCode)(applicableTo) → (promoDesc)(createdBy)(discUnit)
                                                    (discRate)(startDate)(endDate)}
```

Contains schema : BCNF

```
CREATE TABLE Contains (
    orderId         INTEGER REFERENCES Orders
                                ON DELETE CASCADE
                                ON UPDATE CASCADE,
    rname           VARCHAR(100),
    fname           VARCHAR(100),
    foodQty         INTEGER NOT NULL,
    reviewContent   VARCHAR(300),
    PRIMARY KEY(orderId, rname, fname),
    FOREIGN KEY(rname, fname) REFERENCES Sells(rname, fname),
    CHECK(foodQty >= 1)
);

Non-trivial FDs F = {(orderId)(rname)(fname) → (foodQty)(reviewContent)}
```

Orders schema : BCNF

```
CREATE TABLE Orders (
    orderId             INTEGER,
    userId              INTEGER NOT NULL REFERENCES Customers ON DELETE CASCADE ON UPDATE CASCADE,
    promoCode           VARCHAR(20),
    applicableTo        VARCHAR(200),
    modeOfPayment       VARCHAR(10) NOT NULL,
    timeOfOrder         TIMESTAMP NOT NULL,
    deliveryLocation    VARCHAR(100) NOT NULL,
    usedRewardPoints    INTEGER DEFAULT 0,
    givenRewardPoints   INTEGER NOT NULL,
    PRIMARY KEY(orderId),
    FOREIGN KEY(promoCode, applicableTo)  REFERENCES Promotions,
    CHECK(modeOfPayment = 'cash' OR
          modeOfPayment ='credit'),
    CHECK (usedRewardPoints = 5 OR
            usedRewardPoints = 10 OR
            usedRewardPoints = 15 OR
            usedRewardPoints = 0)
);

Non-trivial FDs F = {orderId → (userId)(promoCode)(applicableTo)
                               (modeOfPayment)(timeOfOrder)(deliveryLocation)
                               (usedRewardPoints)(givenRewardPoints)}
```

Delivers schema : BCNF

```
CREATE TABLE Delivers (
    orderId                  INTEGER REFERENCES Orders
                                    ON DELETE CASCADE
                                    ON UPDATE CASCADE,
    userId                   INTEGER NOT NULL,
    departTimeForRestaurant  TIMESTAMP,
    departTimeFromRestaurant TIMESTAMP,
    arrivalTimeAtRestaurant  TIMESTAMP,
    deliveryTimetoCustomer   TIMESTAMP,
    rating                   INTEGER,
    PRIMARY KEY (orderId),
    FOREIGN KEY (userId) REFERENCES Riders
                            ON DELETE CASCADE,
    CHECK(rating <= 5)
);

Non-trivial FDs F = {(orderId) → (userId)(departTimeForRestaurant)(departTimeFromRestaurant)
                                  (arrivalTimeAtRestaurant)(deliveryTimetoCustomer)(rating)}
```

MinSpendingPromotions schema : BCNF

```
CREATE TABLE MinSpendingPromotions (
    promoCode       VARCHAR(20),
    applicableTo    VARCHAR(200),
    minAmt          NUMERIC(8, 2) DEFAULT 0,
    PRIMARY KEY (promoCode, applicableTo),
    FOREIGN KEY (promoCode, applicableTo) REFERENCES Promotions
                                            ON DELETE CASCADE
                                            ON UPDATE CASCADE
);

Non-trivial FDs F = {(promoCode)(applicableTo) → minAmt}
```

CustomerPromotions schema : BCNF

```
CREATE TABLE CustomerPromotions (
    promoCode               VARCHAR(20),
    applicableTo            VARCHAR(200),
    minTimeFromLastOrder    INTEGER, -- # of days
    PRIMARY KEY (promoCode, applicableTo),
    FOREIGN KEY (promoCode, applicableTo) REFERENCES Promotions
                                            ON DELETE CASCADE
                                            ON UPDATE CASCADE
);

Non-trivial FDs F = {(promoCode)(applicableTo) → minTimeFromLastOrder}
```

4.1. Constraints not captured by Relational Schema

Intervals - For the same rider, no intervals should overlap with one another. There must be at least 1 hour of break between any 2 consecutive intervals. Intervals must fall within the start and end date of the WWS they belong to.

Weekly Work Schedule - For each worker, there should be no overlapping WWS. Each WWS must be at least 10 hours and at most 48 hours in total. Each WWS must be declared for exactly 7 consecutive days.

Monthly Work Schedule - For each week in of the MWS, the 4 comprising WWS must be equivalent. Each WWS should have 5 consecutive work days, that comprise of intervals using the pre-defined shifts for full-time riders. Each MWS should last for 28 days exactly, and there should not be any overlapping MWS for the same rider.

Riders - During the operation hours of the FDS, there should be at least five riders (part-time or full-time) working at each hourly interval.

Orders - Quantity of food ordered for a particular food item cannot exceed it’s availability. Total cost order must hit a certain minimum order amount set by the restaurant.

Delivers - Assignment of rider has to ensure that the rider is currently on his work shift, and is free to deliver the order.

5. Triggers

5.1. Ensure No Overlap Between Intervals

Trigger: interval_overlap_trigger

This trigger makes sure that within a same schedule which belongs to only one rider, there must not exist an overlap of different intervals. This is a different implementation from the `OVERLAPS` operator provided by PSQL. The `OVERLAPS` operator does not consider intervals with a single common endpoint to overlap but our implementation does, in order to better fit our use case.

This is done by ensuring:

- For two intervals belonging to the same schedule and falls on the same date, they do not have any properties of a wrong input schedule.

```
CREATE OR REPLACE FUNCTION check_intervals_overlap_deferred() RETURNS TRIGGER AS
$$
DECLARE
    badInputSchedule INTEGER;
BEGIN
    SELECT DISTINCT I1.scheduleId
    INTO badInputSchedule
    FROM Intervals I1
    WHERE EXISTS(
                    SELECT 1
                    FROM Intervals I2
                    WHERE I2.scheduleId = I1.scheduleId
                      AND I2.intervalId <> I1.intervalId
                      AND I2.startTime::date = I1.startTime::date
                      AND (
                            (I2.startTime::time <= I1.startTime::time
                                AND I2.endTime::time >= I1.startTime::time)
                            --IE: I2 is 2-5pm , I1 is 3 - 4pm / 3 - 6pm
                            OR
                            (I2.startTime::time <= I1.endTime::time
                                AND I2.endTime::time >= I1.endTime::time)
                            --IE: I2 is 2-5pm, I1 is 12pm - 3pm / 12pm - 6pm
                            OR (
                                        DATE_PART('hours', I1.startTime) - DATE_PART('hours', I2.endTime) <
1
                                    AND DATE_PART('hours', I1.startTime) >= DATE_PART('hours', I2.endTime)
                              -- IE: I1 : 3-5pm, I2 is 11am - 2.30pm  (this constraint of one hour
difference is also capture in schema)
                                    )
                        )
                );
    IF badInputSchedule IS NOT NULL THEN
        RAISE EXCEPTION 'scheduleId % has some overlapping intervals', badInputSchedule;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE PLPGSQL;


CREATE CONSTRAINT TRIGGER interval_overlap_trigger
    AFTER INSERT
    ON Intervals
    DEFERRABLE INITIALLY DEFERRED
    FOR EACH ROW
EXECUTE FUNCTION check_intervals_overlap_deferred();
```

5.2. Ensure 5 Consecutive Days In Each WWS of a MWS

Trigger: mws_5days_trigger

This is to enforce the constraint of “Each WWS in a MWS must consist of five consecutive work days”. This is done by ensuring:

- there are 5 distinct days that can be obtained from the work intervals for each week
- the difference between the first interval and the last interval of work is 4 days , i.e. all intervals fall within 5 days.

```
DROP FUNCTION IF EXISTS check_mws_5days_consecutive_constraint_deferred() CASCADE;
CREATE OR REPLACE FUNCTION check_mws_5days_consecutive_constraint_deferred() RETURNS TRIGGER AS
$$
DECLARE
    lastIntervalStartTime  TIMESTAMP;
    firstIntervalStartTime TIMESTAMP;
    distinctDates          INTEGER;
BEGIN
    WITH curr_Intervals AS (
        SELECT *
        FROM Intervals I
        WHERE I.scheduleId = NEW.scheduleId1
    )
    SELECT startTime
    into lastIntervalStartTime
    FROM curr_Intervals I
    ORDER BY endTime DESC
    LIMIT 1;

    WITH curr_Intervals AS (
        SELECT *
        FROM Intervals I2
        WHERE I2.scheduleId = NEW.scheduleId1
    )
    SELECT startTime
    into firstIntervalStartTime
    FROM curr_Intervals I
    ORDER BY endTime ASC
    LIMIT 1;

    WITH curr_Intervals AS (
        SELECT *
        FROM Intervals I3
        WHERE I3.scheduleId = NEW.scheduleId1
    )
    SELECT COUNT(DISTINCT I.startTime::date)
    into distinctDates
    FROM curr_Intervals I;
    IF ((lastIntervalStartTime::date - firstIntervalStartTime::date) <> 4 --all intervals within 5 days
        OR distinctDates <> 5) -- each day got interval
    THEN
        RAISE EXCEPTION 'MWS must have 5 consecutive work days';
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE PLPGSQL;

CREATE CONSTRAINT TRIGGER mws_5days_trigger
    AFTER INSERT
    ON Monthly_Work_Schedules
    DEFERRABLE INITIALLY DEFERRED
    FOR EACH ROW
EXECUTE FUNCTION check_mws_5days_consecutive_constraint_deferred();
```

5.3. Ensure MWS Intervals Follow Predefined Timing

Trigger: mws_predefined_interval_trigger

This trigger enforces each work day to only consist of the predefined intervals for full-time riders (e.g. 10pm to 2pm, 3pm to 7pm). This is achieved by pairing intervals belonging to the same day and scheduleId through joining 2 Interval instances using the startTime of the intervals. Then, we check:

- Each interval declared has a corresponding pair, that is 1 hour apart, and are both 4 hours long
- The first interval starts at 10am/11am/12pm/1pm

```
DROP FUNCTION IF EXISTS check_mws_intervals_constraint_deferred() CASCADE;
CREATE OR REPLACE FUNCTION check_mws_intervals_constraint_deferred() RETURNS TRIGGER AS
$$
DECLARE
    badInputSchedule INTEGER;
BEGIN
    WITH curr_Intervals AS (
        SELECT *
        FROM Intervals I
        WHERE I.scheduleId = NEW.scheduleId1
    ),
         Interval_Pairs (intervalId1, startTime1, endTime1, intervalId2, startTime2, endTime2) AS (
             select cI1.intervalId, cI1.startTime, cI1.endTime, cI2.intervalId, cI2.startTime,
cI2.endTime
             from curr_Intervals cI1,
                  curr_Intervals cI2
             where cI1.startTime::date = cI2.startTime::date -- 2 intervals of the same day
               and cI1.startTime::time < cI2.startTime::time -- cI1 is the earlier timing, cI2 the later
         )
    SELECT S.scheduleId
    INTO badInputSchedule
    FROM Weekly_Work_Schedules S
    WHERE S.scheduleId = NEW.scheduleId1
      AND ( NOT EXISTS( -- table is non-empty
                    select 1 from Interval_Pairs IP2 limit 1
                )
            OR EXISTS( --checks for any bad intervals
                    SELECT 1
                    FROM Interval_Pairs IP
                    WHERE (select count(*) from Interval_Pairs) <>
                          ((select count(*) from curr_Intervals) / 2) -- each interval has a pair
                       OR NOT (
                            IP.startTime1::time = '10:00' OR
                            IP.startTime1::time = '11:00' OR
                            IP.startTime1::time = '12:00' OR
                            IP.startTime1::time = '13:00'
                        )
                       OR NOT (DATE_PART('hours', IP.endTime1) - DATE_PART('hours', IP.startTime1) = 4
                        AND DATE_PART('hours', IP.endTime2) - DATE_PART('hours', IP.startTime2) = 4)
                       OR NOT (DATE_PART('hours', IP.startTime2) - DATE_PART('hours', IP.endTime1) = 1)
                )
        );

    IF badInputSchedule IS NOT NULL THEN
        RAISE EXCEPTION '% violates some timing in Intervals', badInputSchedule;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE PLPGSQL;

CREATE CONSTRAINT TRIGGER mws_predefined_interval_trigger
    AFTER INSERT
    ON Monthly_Work_Schedules
    DEFERRABLE INITIALLY DEFERRED
    FOR EACH ROW
EXECUTE FUNCTION check_mws_intervals_constraint_deferred();
```

6. Complex Queries

6.1. View Monthly Salary for Rider

This query calculates the amount of salary that a rider (part-time or full-time) receives for the month. First we find the detailed schedule of the rider by joining the interval table with the weekly schedule table.

The salary calculation is as such:

Base salary + Bonus salary. Base salary is calculated by: Number of hours worked * Rate per hour. Rate per hour is determined by whether the rider is a part-timer ($2/h) or full timer($5/h).

Bonus salary is calculated by: Number of deliveries($4 per delivery) made during peak hour (Between the periods of 12:00 - 13:00 and 18:00 - 20:00) + Number of deliveries($2 per delivery) made during non-peak hour.

```
 router.get('/viewMonthSalary', (req, res) => {
    const userId = req.body.userId;
    const month = req.body.month;
    const year = req.body.year;
    const text = `
    with result as (
        select startTime, endTime, date_part('hours', endTime) - date_part('hours', startTime) as
duration
        from Weekly_Work_Schedules S join intervals I
        on (S.scheduleId = I.scheduleId)
        and (S.userid = $1) and (SELECT EXTRACT(MONTH FROM S.startDate::date)) = $2
        and (SELECT EXTRACT(YEAR FROM S.startDate::date)) = $3),
    result2 as (
        SELECT D.deliveryTimetoCustomer, case
                                        when ((deliveryTimetoCustomer::time >= '12:00' and
deliveryTimetoCustomer::time <= '13:00')
                                                OR (deliveryTimetoCustomer::time >= '18:00' and
deliveryTimetoCustomer::time <= '20:00'))
                                        then 4
                                        else 2
                                        end as delivery_fee
        FROM Delivers D
        WHERE userId = $1
        AND (SELECT EXTRACT(MONTH FROM D.deliveryTimetoCustomer::date)) = $2
        AND (SELECT EXTRACT(YEAR FROM D.deliveryTimetoCustomer::date)) = $3),
    result3 as (
        select coalesce((select sum(duration) from result R),0) as totalHoursWorked ,
coalesce(sum(delivery_fee),0) as totalFees
        from result2 R2)
    select R3.totalHoursWorked, R3.totalFees, case
        when $1 not in (select PT.userId from Part_Time PT) then (R3.totalHoursWorked * 5 + totalFees)
        else (R3.totalHoursWorked * 2 + totalFees) --part_time
        end as pay
    from result3 R3
    `;

    const values = [userId, month, year];
    pool
        .query(text, values)
        .then(result => {
            console.log(result.rows);
            res.json(result.rows);
        })
        .catch(e => console.error(e.stack))
})
```

6.2. Check current status for Rider

This is a complex query to find the current status of the rider. All riders will be in three states:

- Rider is not working
- Rider is working and free to accept orders.
- Rider is working and currently delivering orders.

In the main function, we are able to identify the status of the rider by checking whether he is working or he is on his break at this current time. If he is working, we would check again to see if he is currently delivering or not by checking the expected delivery time to the customer:

```
CREATE OR REPLACE FUNCTION findStatusOfRider(riderId INTEGER, current TIMESTAMP)
    RETURNS INTEGER AS
$$
DECLARE
    latestDelivery TIMESTAMP;
    result         INTEGER;

BEGIN
    SELECT D.deliveryTimetoCustomer
    INTO latestDelivery
    FROM Delivers D
    WHERE D.userId = riderId
    ORDER BY D.deliveryTimetoCustomer desc
    LIMIT 1;

    IF latestDelivery IS NULL THEN
        latestDelivery = '1970-01-01 00:00:00';
    END IF;

    CASE
        WHEN checkWorkingStatusHelperOfRider(riderId, current) = 0 then result = 0;
        WHEN latestDelivery < current THEN result = 1;
        WHEN current <= latestDelivery THEN result = 2;
        ELSE result = -1;
        END CASE;
    RETURN result;
END;
$$ LANGUAGE PLPGSQL;
```

This is facilitated by a helper function where we first find out their detailed schedule through a helper function that joins the schedule table and interval table: (see next page)

```
CREATE OR REPLACE FUNCTION checkWorkingStatusHelperOfRider(riderId INTEGER, current TIMESTAMP)
    RETURNS INTEGER AS
$$
DECLARE
    currentDate DATE;
    currentTime TIME;
    result      INTEGER;

BEGIN
    currentTime = current::time;
    currentDate = current::date;

    CASE
        WHEN EXISTS(
                SELECT 1
                FROM Intervals I
                WHERE I.startTime::time <= currentTime
                  AND I.endTime::time > currentTime
                  AND I.startTime::date = currentDate
                  AND I.scheduleId = (SELECT W.scheduleId
                                      FROM Weekly_Work_Schedules W
                                      WHERE W.startDate::date <= currentDate
                                        AND W.endDate::date >= currentDate
                                        AND W.userId = riderId)
            ) THEN result = 1;
        ELSE result = 0;
        END CASE;
    RETURN result;
END;
$$ LANGUAGE PLPGSQL;
```

7. Software tools / Frameworks

Frontend : React.js

Platform : Node.js

Framework : Express.js

Database : PostgreSQL

Version Control : Git & GitHub

Code Editor: IntelliJ IDEA

Languages used

- Javascript
- SQL for database
- Python for generation of large dataset

8. Application Screenshot

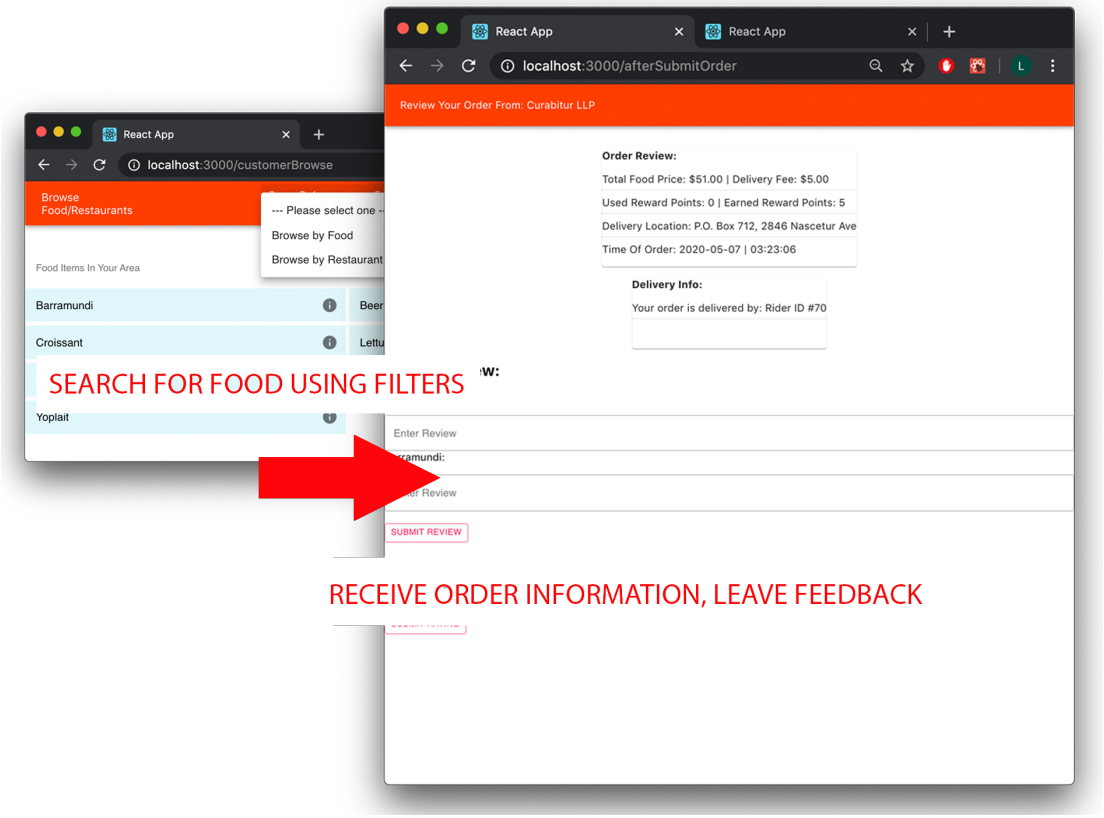

Figure 2. Customer can browse for food, make an order, and make a review

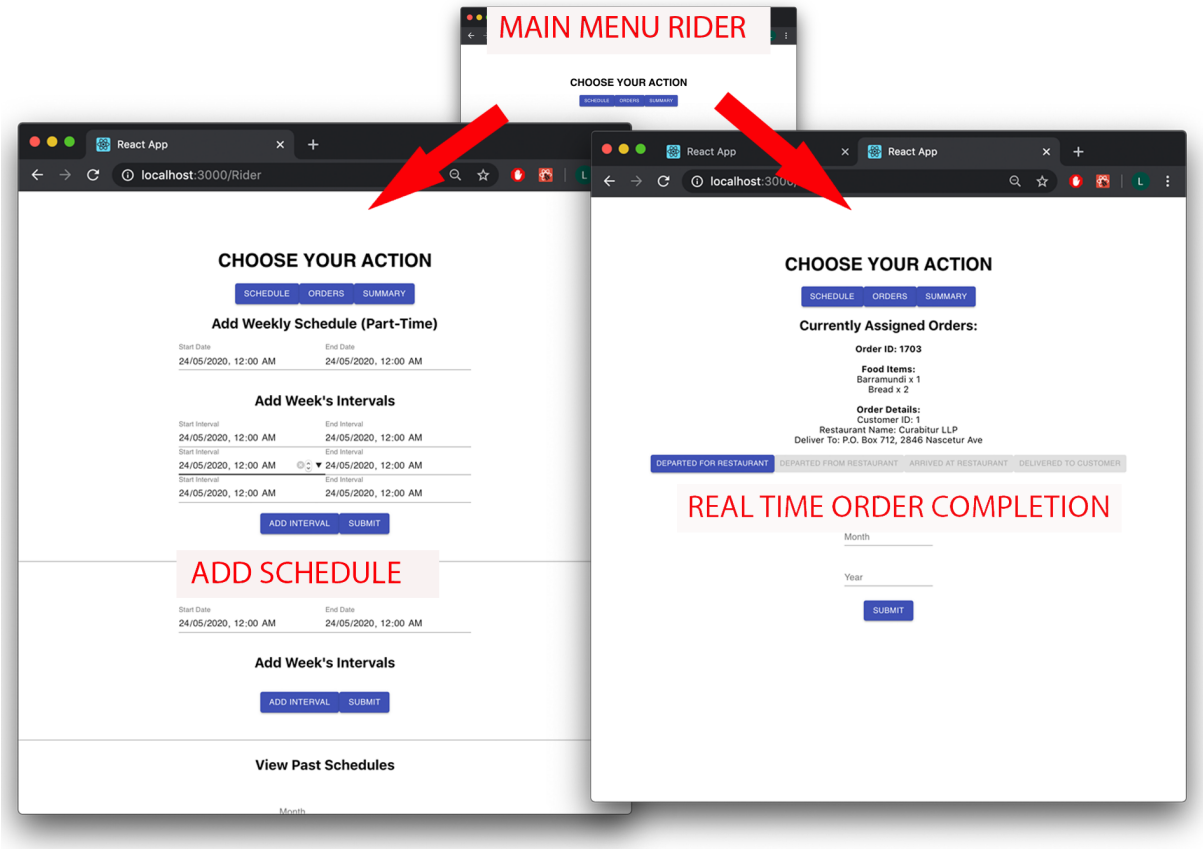

Figure 3. Rider can declare schedule, and complete in order real-time

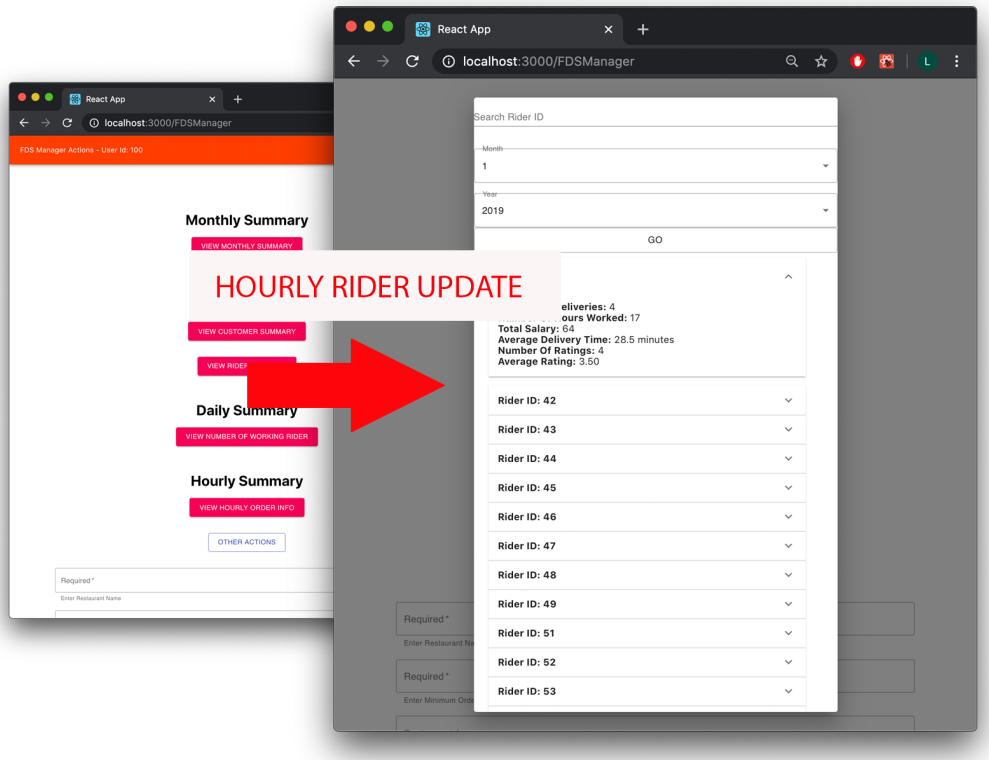

Figure 4. FDS managers can view all summary info (e.g. Hourly rider update)

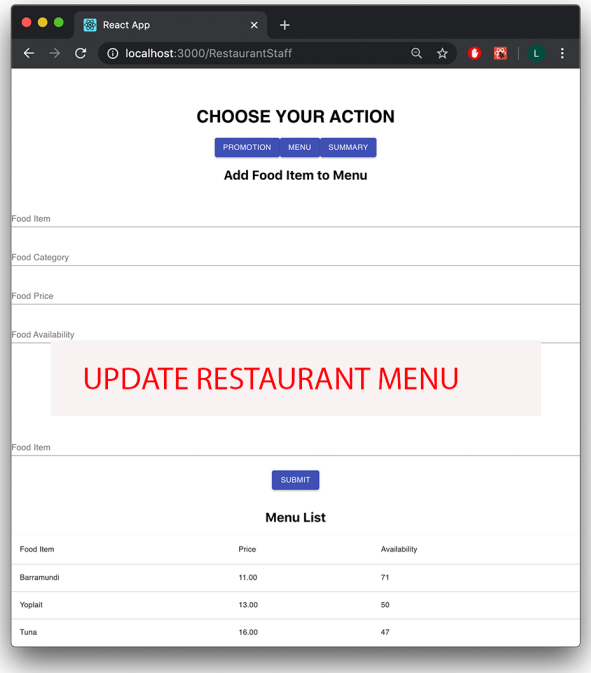

Figure 5. Restaurant Staff can update menu dyanmically

9. Conclusion/Summary

In summary, this project served as a very challenging learning experience for us. All of us were relatively new to full stack development and using technologies like PERN (PostgresSQL, Express, Node.js and React). As such, we had to research a lot to find out how each of these technology work and how to integrate them to come up with an application. At times, we were also unsure of what the best practices were when building the application. For example, we were uncertain if we should implement a particular logic in the backend or if we should handle it in the database as both ways are possible.

Through these challenges, aside from learning modern technologies, we learnt the importance of thinking from the perspective of the database when making an application, and the important factors when designing a database, such as good schema design. We saw firsthand the gradual improvement of our schema design as we were introduced to concepts such as normal forms. We learnt how to critically evaluate our system before implementation and afterwards continuously better meet required specifications by incorporating a design thinking iterative process.