Campus Amenities Administration

CS355 Mini Project - Group #2

Members: Vaishakh (1901CS68) | Varsha Tumburu (1901CS69) | Venkatadri Arava (1901CS70)

PURPOSE

Every administration needs to keep track of many services for their users. With the number of students rising year over year, it is the need of the hour to automate the tons of paperwork, numerous repetitive workflows, and data. We need to rely on a system that can perform some, if not most, of the tasks even faster and more accurately than a human can. In this project, we develop a working prototype to satisfy the below-stated requirements.

SCOPE

Although various services within the institution can be taken up, we will be working on the following amenities: Guesthouse, Market shop, and Landscaping.

Guesthouse Services

In the Guesthouse, there are mainly 4 users. The general user, the cooks, the supervisor and the guest house staff or manager. The general user can book rooms, order food and view their bills. The cooks can see the bills live, they can also modify the menu. The supervisor can do the scheduling for the employees. The guest house manager can generate bills in PDF format and can also maintain the guest house expenditures.

Market Shop Services

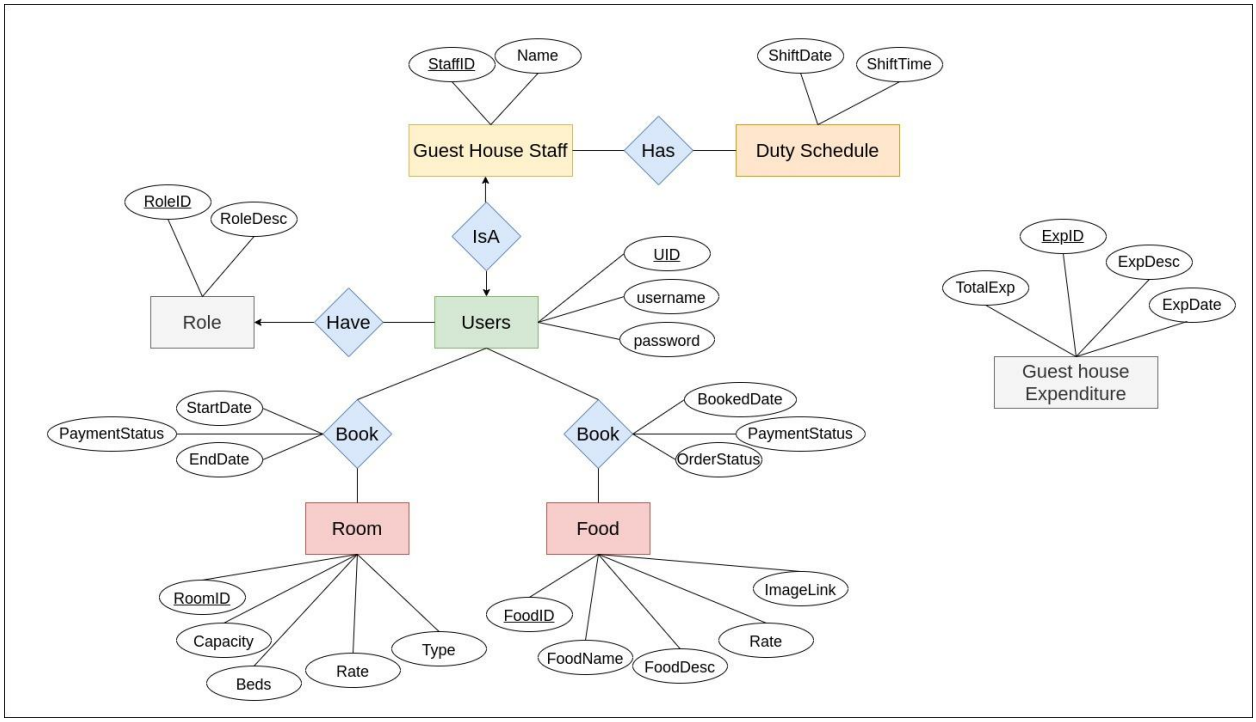
It includes functionalities like maintaining records of shops and respective shopkeepers. We will also need to keep track of the store's license period, any extension period granted, monthly rent/ electricity payment details. And most importantly, we require a mechanism to acquire customer feedback and their expectations from the market.

Landscaping Services

In landscaping services, there are mainly 3 users namely: Gardener, Supervisor and Vendor. The system contains functionalities such as Scheduling Weekly Timetables for Gardening, allowing Gardeners to take Notes, allowing Vendors to request payment, allowing normal users to put in grass cutting requests and so on.

GUEST HOUSE SERVICES

ER Diagram



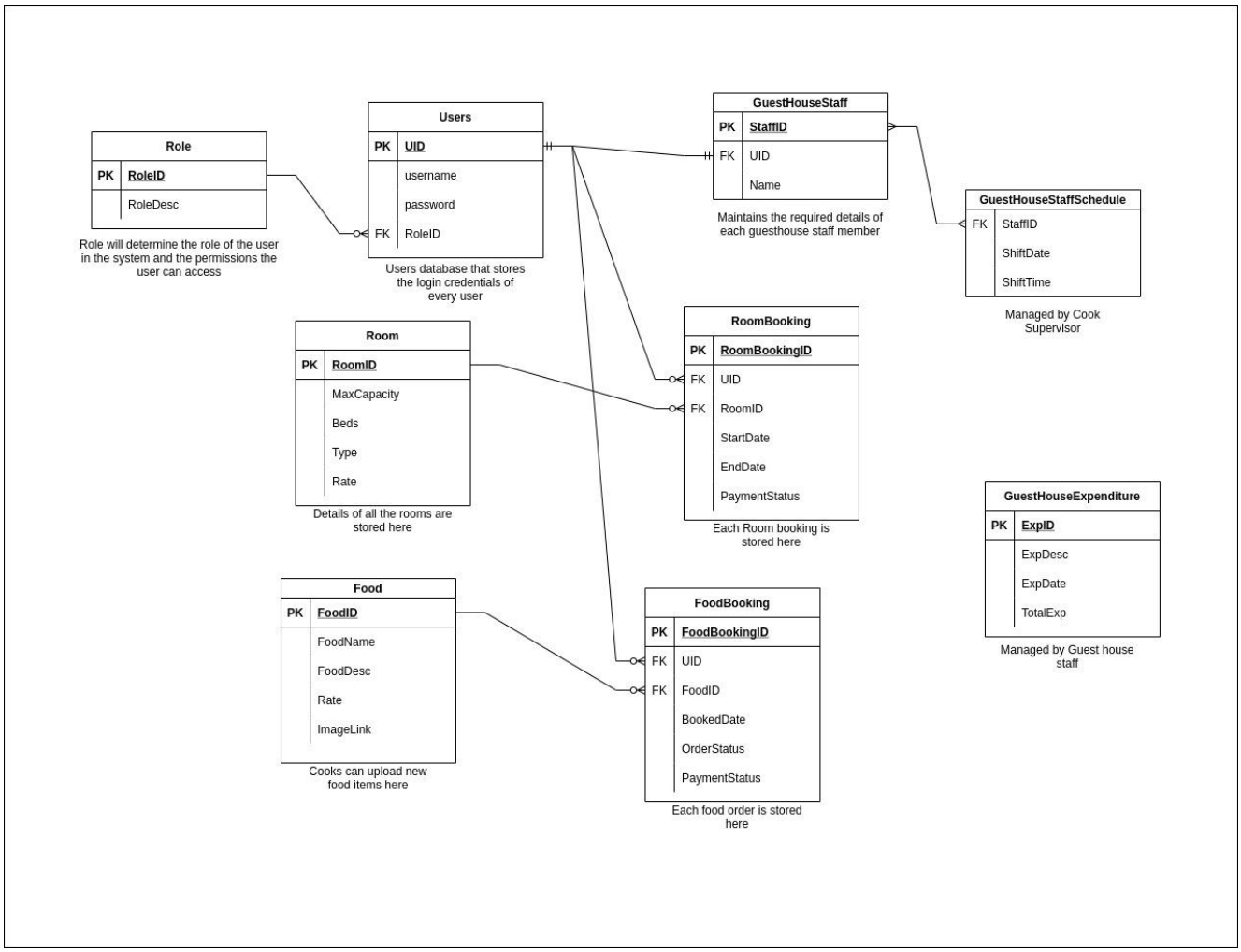
Tables and Description:

The various tables in the system are:

- **Users:** Maintains the records of all users in the system. It contains **UID** (User ID), username and password of the user.
- **Roles:** Maintains the details of each Role. A role will grant certain permissions to the User. It stores **RoleID** and Role Description.
- **Room:** Maintains the details of each room. It contains **RoomID** and other details like Capacity, Number of Beds, type of room and price per day of stay is stored.
- **Food:** Details about food, like Food Name, Food description and Rate of food is stored here.
- **GuestHouseStaff:** Details about each staff member like **StaffID** and name is stored here.
- **GuestHouseExpenditure:** Stores all the expenditures incurred by the guesthouse.
- **RoomBooking:** Stores the details of all the room bookings. It stores the User ID of the client who has booked a room, the room ID, and the dates of stay (Start date to End date). It also stores the Payment Status, which shows whether a user has paid or not.
- **FoodBooking:** Stores the details of all the food orders. It stores the User ID of client, the Food ID and the date of order. It also stores the Payment Status, which shows whether a

user has paid or not and the Order Status which shows if the Order has been delivered or not.

- **GuestHouseStaffSchedule:** Stores the schedule of the cooks in the guest house. It stores staff ID and the Shift date as well as Shift time. It can be queried over a set of dates to view the entire schedule of a month.



Essential Queries and codes

Registration

During registration, the user enters a username, password and the role for the user. The roles for Guesthouse service comprises all roles with RoleID between 200 and 299 and the RoleID ‘2’. Hence for each user with such a role ID, we need a trigger to automatically add them to the staff

table. The Trigger used for this is:

```
DELIMITER $$
CREATE TRIGGER add_guesthouse_employee
AFTER INSERT
ON Users FOR EACH ROW
BEGIN
    IF (NEW.RoleID = 2) OR (NEW.RoleID>200) THEN
        INSERT INTO GuestHouseStaff VALUES (NULL,NEW.UID,NEW.username);
    END IF;

END $$
DELIMITER ;
```

Room Booking

For room booking, I am first verifying if a room is available or not using the following code:

```
const [checkQuery, checkFields] = await db.query(
  "Select RoomID FROM `RoomBooking` WHERE ((StartDate >= ? AND StartDate <= ?) OR (EndDate >= ? AND EndDate <= ?)) AND RoomID=?",
  [startDate, endDate, startDate, endDate, RoomID]
);

if (checkQuery.length > 0) {
  return res.send({
    success: false,
    message: "Room is already booked in given date",
  });
}
```

If this is satisfied, I will insert the values into the table

```
const UIDQuery = "SELECT `UID` FROM `Users` WHERE `username` = ?";

const [q1rows, q1fields] = await db.query(UIDQuery, [username]);
const UID = q1rows[0].UID;

const [rows, fields] = await db.query(
  "INSERT INTO `RoomBooking` VALUES (NULL,?,?,?,?,'Pending')",
  [UID, RoomID, startDate, endDate]
);

return res.send({
  success: true,
});
```

Food Booking

For food booking, I will simply insert an entry into the FoodBooking table using the following code:

```
await db.query(
  "INSERT INTO `FoodBooking` VALUES (NULL,?,?,?,'Placed','Pending')",
  [uid[0].UID, FoodID, datetime.toISOString().slice(0, 10)]
);
```

Bill Generation

For bill generation I have created two views.

Using the following code:

```
CREATE VIEW RoomBills AS(
SELECT UID, RoomID, EndDate, (Days*Rate) AS Cost FROM(
Select RoomBooking.UID, RoomBooking.RoomID, DATEDIFF(EndDate,StartDate) As Days, Room.Rate, RoomBooking.EndDate FROM RoomBooking,
Room WHERE Room.RoomID = RoomBooking.RoomID AND RoomBooking.PaymentStatus='Pending') AS T);



CREATE VIEW FoodBills AS(
SELECT UID, FoodName, BookedDate, Rate FROM
(SELECT FoodBooking.UID, FoodBooking.FoodID, FoodBooking.BookedDate, Food.Rate, Food.FoodName FROM FoodBooking, Food WHERE
PaymentStatus='Pending' AND FoodBooking.FoodID = Food.FoodID) AS T
);
```

We then combine the result of the two views to generate a PDF Bill like so:

Food Bills

Bills for all the Food Bookings

User ID	Food Name	Date of booking	Cost
21	Food3	Mon Nov 29	150
21	Naan	Mon Nov 29	15

Room Bills

Bills for all the Room Bookings

User ID	Room ID	Last Date of stay	Cost
21	101	Wed Dec 08	21000

Total Bill

User ID	Grand total
21	21165

Guest house expenditure

They are recorded in the GuestHouseExpenditure table. The code used for this is:

```
app.post("/api/guesthouse/employee/addExpenditure", async (req, res) => {
  await db.query("INSERT INTO GuestHouseExpenditure VALUES (NULL,?,?,?)", [
    req.body.ExpDesc,
    req.body.ExpDate,
    req.body.TotalExp,
  ]);

  res.send({ success: true });
});
```

We can then perform queries on it to get the monthly results. For this, the code is:

```
app.post("/api/guesthouse/employee/getExpenditure", async (req, res) => {
  const [rows, fields] = await db.query(
    "Select * FROM GuestHouseExpenditure WHERE ExpDate >= ? AND ExpDate <= ?",
    [req.body.startDate, req.body.endDate]
  );

  res.send({ expenditure: rows });
});
```

Staff Scheduling

For staff scheduling, I have implemented **both auto-scheduling and manual scheduling functionalities.** In the case of auto-scheduling, we need 2 contractual cooks, 1 regular cook and 2 helpers, we will then allot them in a fixed fashion for 7 days ahead of a given starting date.

I have inserted into the GuestHouseStaffSchedule table for this. One part of the code used for this is:

```
// Inserting in day 1 - 4
for (let i = 0; i < 4; i++) {
  await db.query(
    "INSERT INTO GuestHouseStaffSchedule VALUES(?,DATE_ADD(?, INTERVAL ? DAY),?), (?,DATE_ADD(?, INTERVAL ? DAY),?)",
    [
      req.body.contractual1.StaffID,
      req.body.startDate,
      String(i),
      "Morning",
      req.body.contractual1.StaffID,
      req.body.startDate,
      String(i),
      "Night",
    ]
  );

  await db.query(
    "INSERT INTO GuestHouseStaffSchedule VALUES(?,DATE_ADD(?, INTERVAL ? DAY),?), (?,DATE_ADD(?, INTERVAL ? DAY),?)",
    [
      req.body.contractual2.StaffID,
      req.body.startDate,
      String(i),
      "Morning",
      req.body.contractual2.StaffID,
      req.body.startDate,
      String(i),
      "Night",
    ]
  );

  await db.query(
    "INSERT INTO GuestHouseStaffSchedule VALUES(?,DATE_ADD(?, INTERVAL ? DAY),?), (?,DATE_ADD(?, INTERVAL ? DAY),?)",
    [
      req.body.cleaner1.StaffID,
      req.body.startDate,
      String(i),
      "Morning",
      req.body.cleaner1.StaffID,
      req.body.startDate,
      String(i),
      "Night",
    ]
  );
}
```

This code will do scheduling for the first 4 days after a given start date. Similarly I have scheduled for the rest of the week.

Manual scheduling will overwrite any current schedule and manually add in an entry. One part of the code for this is as follows:

```
await db.query(
  "DELETE FROM GuestHouseStaffSchedule WHERE ShiftDate = ? AND ShiftTime = ?",
  [req.body.startDate, req.body.shift]
);

await db.query("INSERT INTO GuestHouseStaffSchedule VALUES (?,?,?)", [
  req.body.cook1.StaffID,
  req.body.startDate,
  req.body.shift,
]);

await db.query("INSERT INTO GuestHouseStaffSchedule VALUES (?,?,?)", [
  req.body.cook2.StaffID,
  req.body.startDate,
  req.body.shift,
]);

await db.query("INSERT INTO GuestHouseStaffSchedule VALUES (?,?,?)", [
  req.body.cleaner.StaffID,
  req.body.startDate,
  req.body.shift,
]);
```

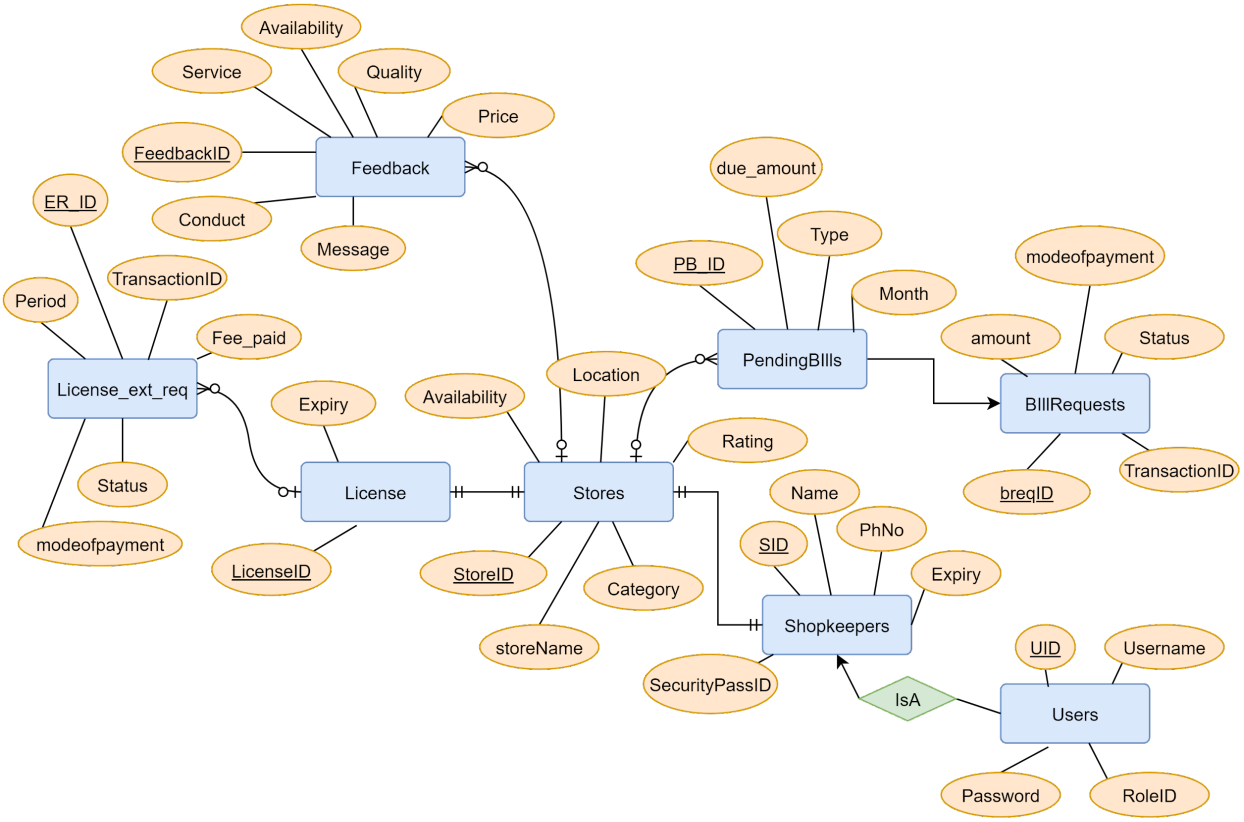
As you can see, we are deleting any current entry and then writing in new values.

Other Functionalities:

- Cook can see orders live. I have used a polling mechanism to update the orders table every 10 seconds. The cook can also deliver food items and mark an order as delivered.
- Cook can add a new food item with a photo. For this, I am storing the image path in the database and I am using the path to fetch the image file and render it on screen.
- Users can see their pending bills. In case of Food bills, the bills will be displayed **only** for food items which are delivered to them.
- Cooks can see the schedule from a start date to an end date.
- Managers can manage the bills, they can manually clear each bill (Bill of each food item) or they can clear the entire bill of a user too. Both have been implemented

MARKET SHOP SERVICES

ER Diagram



Tables & Utilities

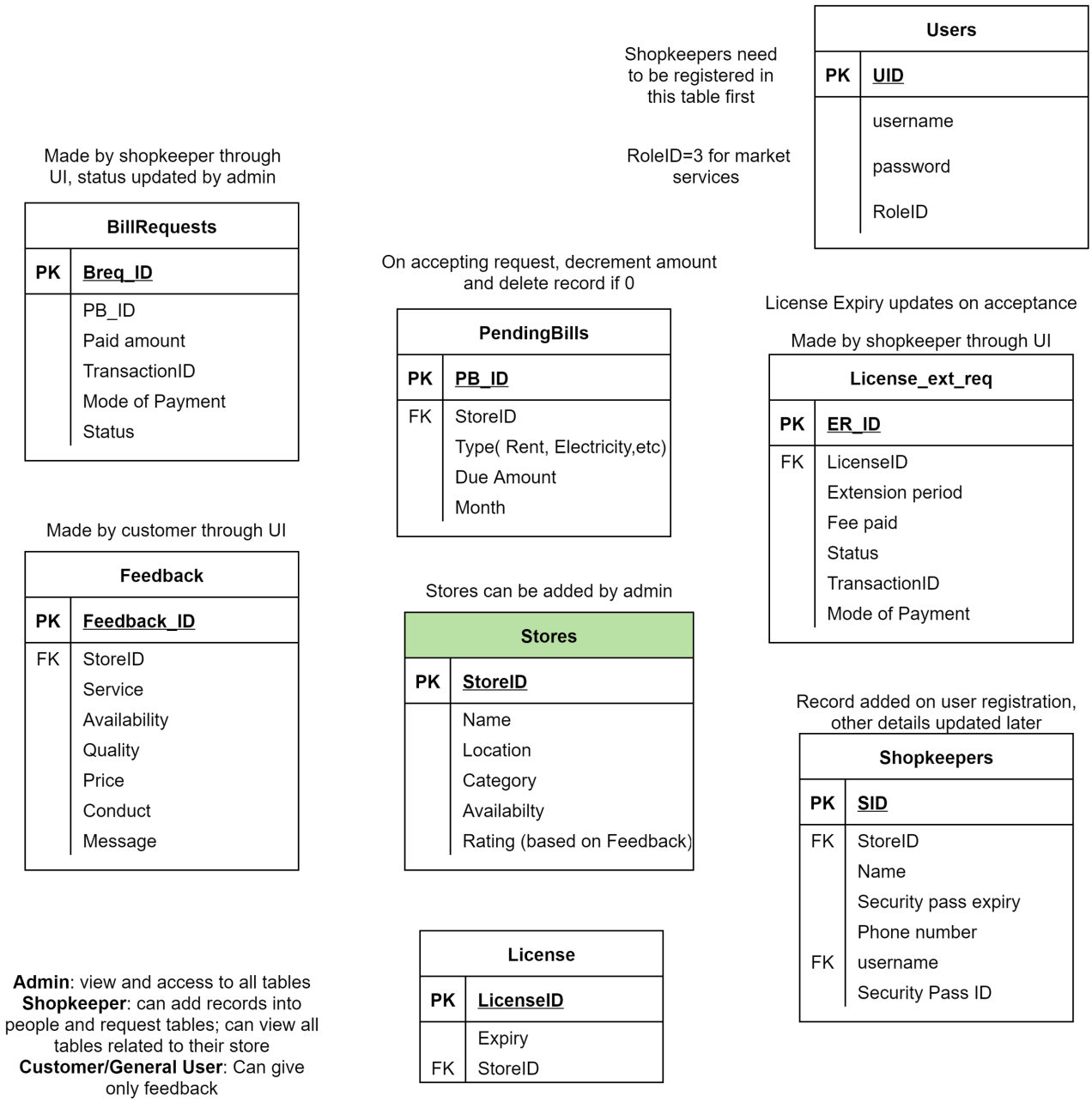
The below diagram shows all the tables used in this section along with respective primary keys, foreign keys and user privileges.

Stores is our main table which has all the details of all the shops on campus from cafes to vegetable markets. This table can be updated only by the admin but can be viewed by the respective shop owner. **Shopkeepers** is another useful table that stores all the personal information of the owner. For simplicity's sake, one shopkeeper can own only one shop. A registration under the market services role triggers a new record in this table. An alert message pops up if the security pass expiry is within a month.

License has all details regarding shop licensing. Each shop has one License ID. On nearing the expiry date, a similar pop-up is used to alert the shopkeeper. The license can be extended by placing a request to the admin after paying the corresponding fee through **License_ext_req.** Once approved by the admin, the expiry date will be updated.

PendingBills can only be updated by the admin but can be viewed by the shopkeeper. A shopkeeper can request the verification of a bill through **BillRequests** so that it is updated in the system and further notices aren't sent. An approved bill request deletes the record from PendingBills if paid amount = due amount. Otherwise, it subtracts paid from due and the record still remains. Notice PB_ID is not a foreign key in BillRequests, since an approved request can delete that record from PendingBills.

Feedback provides a mechanism for the customer (general user) to submit their responses through a form which are later reflected on the store profile for the shopkeeper. An average is taken from this table to calculate the rating in Stores.



Queries, Procedures and Triggers

To execute the above-mentioned functionalities, the following queries have been used:

```
app.get("/api/extractStores", async (req, res) => {
  const query = "SELECT * from `Stores`";
  const [rows, fields] = await db.query(query);

  try {
    res.send({ success: true, info: rows });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});
```

The left side query is used to extract all the stores present in the database to assign the shopkeeper one.

getStoreDetails allows the shop owner to view all the information related to the store from the location to license details. (2 queries on 2 tables)

```
app.get("/api/getStoreDetails", async (req, res) => {
  const query =
    "SELECT * from `Stores` WHERE `StoreID`=(SELECT `StoreID` FROM `Shopkeepers` WHERE `username`=?) ";
  const [rows, fields] = await db.query(query, [req.session.username]);

  const lquery =
    "SELECT * from `License` WHERE `StoreID`=(SELECT `StoreID` FROM `Shopkeepers` WHERE `username`=?) ";
  const [lrows, lfields] = await db.query(lquery, [req.session.username]);

  var obj = Object.assign({}, rows[0], lrows[0]);
  obj.LicenseExpiry = obj.LicenseExpiry.toDateString();
  try {
    res.send({ success: true, info: obj });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});
```

```
app.get("/api/getPendingBills", async (req, res) => {
  const query =
    "SELECT * from `pendingbills` WHERE `StoreID`=(SELECT `StoreID` FROM `Shopkeepers` WHERE `username`=?) ";
  const [rows, fields] = await db.query(query, [req.session.username]);

  try {
    res.send({ success: true, info: rows });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});
```

getPendingBills allows the shopkeeper to view all the bills that he hasn't paid yet. Since username cannot be directly used, another table is used in between.

```
"SELECT `a`.`pb_id` as `id`,`type`,`status` from `billrequests` `a` LEFT JOIN `pendingbills` `b` ON
`a`.`pb_id`=`b`.`pb_id` WHERE `StoreID` = (SELECT `StoreID` FROM `Shopkeepers` WHERE `username` =?)"
"SELECT `licenseID` as `id`,`status` from `license_ext_req` WHERE `LicenseID`=(SELECT `LicenseID` from
`License` WHERE `StoreID`=(SELECT `StoreID` FROM `Shopkeepers` WHERE `username`=?))"
```

The above 2 queries are used as a part of **getRequestStatus**, which allows transparency in the system by giving the shopkeeper information on the status of their request (license or bills).

```
app.get("/api/getBillRequests", async (req, res) => {
  const query =
    "SELECT `a`.*,`storeID`,`type`,`month`,`due_amount` from `billrequests` `a` LEFT JOIN `pendingbills` `b` ON `a`.`pb_id`=`b`.`pb_id` WHERE `status`!='Accepted'";
  const [rows, fields] = await db.query(query);

  try {
    res.send({ success: true, info: rows });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});

app.get("/api/getLicenseRequests", async (req, res) => {
  const query =
    "SELECT `a`.*,`storeID`,`licenseExpiry` from `license_ext_req` `a` LEFT JOIN `license` `b` ON `a`.`licenseID`=`b`.`licenseID` WHERE `status`!='Accepted'";
  const [rows, fields] = await db.query(query);

  try {
    res.send({ success: true, info: rows });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});
```

getBillRequests and **getLicenseRequests** are used as a part of the admin profile to get all the information regarding the request, to decide whether to approve it or not.

```
app.get("/api/getShopkeeper", async (req, res) => {
  const username = req.session.username;

  const [rows, fields] = await db.query(
    "SELECT * FROM `shopkeepers` WHERE username=?",
    [username]
  );

  try {
    const [rows2, fields2] = await db.query(
      "SELECT `StoreName` FROM `Stores` WHERE `StoreID`=?",
      [rows[0].storeID]
    );

    res.send({
      success: true,
      username: req.session.username,
      name: rows[0].name,
      storeID: rows[0].storeID,
      storeName: rows2[0].StoreName,
      phonenumber: rows[0].phonenumber,
      securitypassID: rows[0].securitypassID,
      passexpiry: rows[0].passexpiry.toDateString(),
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});
```

getShopkeeper retrieves all data related to the shop owner from all personal details to which shop he owns, etc. This is displayed on the shopkeeper's profile after logging in. It also has the security pass expiry which is used to alert the user on nearing the date.

```
app.post("/api/addShopkeeperDetails", async (req, res) => {
  const username = req.session.username;
  const name = req.body.name;
  const store = req.body.store;
  const phno = req.body.phonenumber;
  const securitypassID = req.body.securitypass;
  const expiry = req.body.expiry;
  const storeID = Number(store.split("-")[0]);

  const passQuery =
    "SELECT * FROM `shopkeepers` WHERE `securitypassID` = ? OR `storeID` = ?";
  const [rows, fields] = await db.query(passQuery, [securitypassID, storeID]);
  if (rows.length > 0) {
    res.send({ success: false, message: "User already exists." });
    throw new Error("User exists.");
  }

  const query =
    "UPDATE `shopkeepers` SET `name`=?,`storeID`=?,`phonenumber`=?,`securitypassID`=?,`passexpiry`=? WHERE `username`=?";
  try {
    db.query(
      query,
      [name, storeID, phno, securitypassID, expiry, username],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
```

addShopkeeperDetails updates details of the shop owner. A record is directly created using trigger **insertShopkeeper** once registered in the table 'Users'. Initially, all values are null. **getFeedback** displays messages customers write about the store to the shopkeeper in the store profile. This can be limited.

```
app.get("/api/getFeedback", async (req, res) => {
  const query =
    "SELECT `message` from `feedback` WHERE `StoreID`=(SELECT `StoreID` FROM `Shopkeepers` WHERE `username`=?) AND `message` IS NOT NULL ";
  const [rows, fields] = await db.query(query, [req.session.username]);

  try {
    res.send({ success: true, info: rows });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});
```

```
app.post("/api/addStore", async (req, res) => {
  const name = req.body.storename;
  const location = req.body.location;
  const category = req.body.category;
  const availability = req.body.availability;

  const query =
    "INSERT IGNORE INTO `Stores`(`storeName`,`location`,`category`,`availability`) VALUES(?,?,?,?)";
  try {
    db.query(query, [name, location, category, availability], (err, res) => {
      if (err) throw err;
      console.log("res is", res);
    });
```

addStore can only be used by the admin to introduce a new shop. Only when a new shop is created, an owner can be assigned to it.

addFeedback inserts a new performance record into the feedback table. An average of all the 5 values is calculated for all the rows and then updated into the Store table for each entry. We use a procedure called **updateRating** to execute this.

```
app.post("/api/addFeedback", async (req, res) => {
  const store = req.body.store;
  const storeID = Number(store.split("-")[0]);
  const service = req.body.service;
  const conduct = req.body.conduct;
  const availability = req.body.availability;
  const quality = req.body.quality;
  const price = req.body.price;
  const message = req.body.message;

  try {
    const query =
      "INSERT IGNORE INTO `Feedback`(`storeID`,`service`,`availability`,`quality`,`price`,`conduct`,`message`) VALUES(?,?,?,?,?,?,?)";
    db.query(
      query,
      [storeID, service, availability, quality, price, conduct, message],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );

    const query2 = "CALL updateRating(?);";
    db.query(query2, [storeID], (err, res) => {
      if (err) throw err;
      console.log("res is", res);
    });
```

```
app.post("/api/addBillRequest", async (req, res) => {
  const username = req.session.username;
  const month = new Date(req.body.month);
  const amount = req.body.amount;
  const type = req.body.type;
  const transactionID = req.body.transactionID;
  const modeofpayment = req.body.modeofpayment;

  const query1 = "SELECT `storeID` FROM `Shopkeepers` WHERE `username`=?";
  const [rows1, fields1] = await db.query(query1, [username]);
  const storeID = rows1[0].storeID;

  const query2 =
    "SELECT `PB_ID` FROM `pendingbills` WHERE `type`=? AND MONTH(`month`)=? AND YEAR(`month`)=? AND `StoreID`=?";
  const [rows2, fields2] = await db.query(query2, [
    type,
    month.getMonth() + 1,
    month.getFullYear(),
    storeID,
  ]);
  const pb_id = rows2[0].PB_ID;

  const query =
    "INSERT IGNORE INTO `billrequests`(`pb_id`,`amount`,`transactionID`,`modeofpayment`) VALUES (?,?,?,?)";
  try {
    db.query(
      query,
      [pb_id, amount, transactionID, modeofpayment],
```

addBillRequest is called when the shopkeeper requests the approval of a bill they have paid. This requires us to few details from pendingBills to insert a new row. This is then reflected on the store profile. The default status is ‘In review’.

updateBillRequest allows the admin to update the status of any bill request. There are 4 options namely, ‘In review’, ‘On hold’, ‘Denied’ and Accepted’. On approving the request and the amount is fully paid, it is no more shown in the DOM and it is removed from pendingbills in the store profile. Otherwise, the amount is subtracted from the record in pendingbills and it still appears in the profile.

```
app.post("/api/updateBillRequest", async (req, res) => {
  const records = req.body;
  const query = "UPDATE `billrequests` SET `status`=? WHERE `breqID`=?";
  try {
    records.forEach(async (obj, i) => {
      db.query(query, [obj.status, obj.id], (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      });
      if (obj.status === "Accepted") {
        const query0 =
          "SELECT `amount`,`pb_id` FROM `billrequests` WHERE `breqID`=?";
        const [rows, fields] = await db.query(query0, [obj.id]);
        const query1 =
          "UPDATE `pendingbills` SET `due_amount`=`due_amount`-? WHERE `pb_id`=? ";
        db.query(query1, [rows[0].amount, rows[0].pb_id], (err, res) => {
          if (err) throw err;
        });

        const query2 =
          "SELECT `due_amount` FROM `pendingbills` WHERE `pb_id`=?";
        const [rows2, fields2] = await db.query(query2, [rows[0].pb_id]);
        console.log(rows2[0].due_amount);
        if (rows2[0].due_amount === 0) {
          const query3 = "DELETE FROM `pendingbills` WHERE `pb_id`=?";
          db.query(query3, [rows[0].pb_id], (err, res) => {
            if (err) throw err;
            console.log("res is", res);
```

```
app.post("/api/updateLicenseRequest", async (req, res) => {
  const records = req.body;
  const query = "UPDATE `license_ext_req` SET `status`=? WHERE `er_id`=?";
  try {
    records.forEach(async (obj, i) => {
      db.query(query, [obj.status, obj.id], (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      });
      if (obj.status === "Accepted") {
        const query0 = "SELECT `period` FROM `license_ext_req` WHERE `er_id`=?";
        const [rows, fields] = await db.query(query0, [obj.id]);

        const query1 =
          "UPDATE `license` SET `licenseExpiry`=DATE_ADD(`licenseExpiry`,INTERVAL ? YEAR) WHERE `licenseID`=(SELECT `licenseID` FROM `license_ext_req`
        db.query(query1, [rows[0].period, obj.id]);
      }
    });
    res.send({
      success: true,
      message: "Status updated.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

Similarly**, updateLicenseRequest** updates the status of the license request. On accepting the request, the request is removed from the profile and the licenseExpiry is updated to expiry+extensionPeriod in the license table.

```
app.post("/api/addLicenseExt", async (req, res) => {
  const username = req.session.username;
  const period = req.body.extPeriod;
  const fee = req.body.fee;
  const transactionID = req.body.transactionID;
  const modeofpayment = req.body.modeofpayment;

  const query1 = "SELECT `storeID` FROM `Shopkeepers` WHERE `username`=?";
  const [rows1, fields1] = await db.query(query1, [username]);
  const storeID = rows1[0].storeID;

  const query2 = "SELECT `licenseID` FROM `license` WHERE `StoreID`=?";
  const [rows2, fields2] = await db.query(query2, [storeID]);
  const licenseID = rows2[0].licenseID;

  const query =
    "INSERT IGNORE INTO `license_ext_req`(`licenseID`,`period`,`fee_paid`,`transactionID`,`modeofpayment`) VALUES (?,?,?,?,?)";
  try {
    db.query(
      query,
      [licenseID, period, fee, transactionID, modeofpayment],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Details updated. Redirect to store profile.",
    });
```

addLicenseExt inserts a request done by the shop owner to extend the license. The respective license ID is retrieved by 2 queries and then inserted accordingly.

Trigger to create a new shopkeeper row on registering under market services.

```
DELIMITER $$
CREATE TRIGGER insertShopkeeper
AFTER INSERT
ON Users FOR EACH ROW
BEGIN
IF(NEW.RoleID=3) THEN
INSERT INTO shopkeepers(username) VALUES (new.username);
END IF;
END$$
DELIMITER ;
```

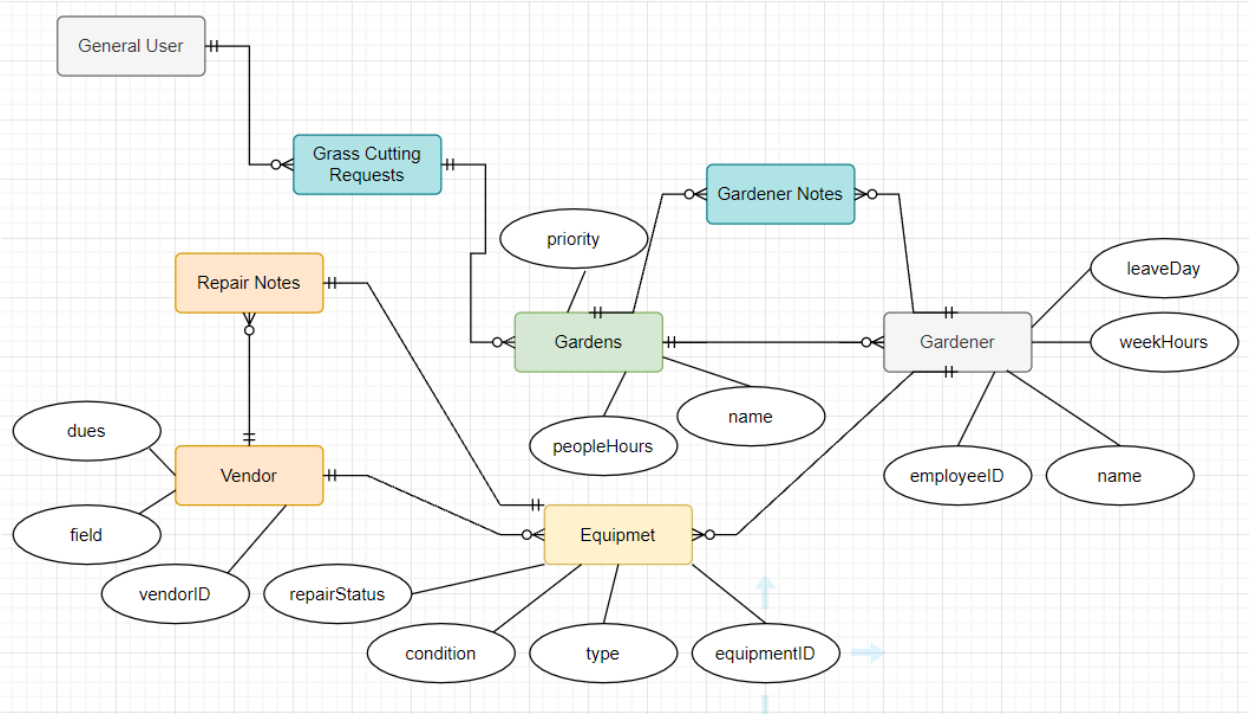
Procedure to update rating in Stores on receiving new feedback for a particular store.

```
DELIMITER $$
CREATE PROCEDURE updateRating(IN sID INT)
BEGIN
DECLARE s,a,q,p,c FLOAT DEFAULT 0;
SELECT AVG(service),AVG(availability),AVG(quality),AVG(price),AVG(conduct) INTO
s,a,q,p,c FROM feedback WHERE storeID=sID;
UPDATE Stores SET rating=((s+a+q+p+c)/5.0) WHERE storeID=sID;
END$$
DELIMITER ;
```

With this, we come to the end of all the API's used in the market services section.

LANDSCAPING SERVICES

ER Diagram



Tables and Description:

There are a total of 15 tables involved in the Landscaping Process. **CutRecs** table, stores all the grass cutting requests put in by users to show the supervisor. **Equipment** table holds basic information about various tools used by the Landscaping services and has **Foreign Keys for Vendor & Gardener**, along with repair status in case it is under repair. **Garden** table is the central table that contains gardens along with attributes like **peopleHours** to measure how long gardeners would have to work and **priority** which is used to schedule. **Gardener** is the table that has a list of working gardeners along with their details such as **workHours, employeeID, leave day** etc. The **vendor** table has the list of vendors and their details, including their pending dues. **The Notes** table contains the Garden Notes written by Gardeners about the various gardens on campus. This table can be read by other gardeners when they work on the respective garden. The **VendorBills** table has a list of payment requests inputted by vendors. The Supervisor can check this list and verify these requests before adding them to their dues. Finally, the **WeekSchedule** is the table that contains the gardening schedule for each week. It can be viewed by gardeners and the Supervisor can manually edit this table if necessary.

Essential Queries, Codes:

Retrieving Gardeners Details: This code is used to take details of the gardener from the database.

```
app.get("/api/getGardener", async (req, res) => {
  console.log("User cookie is", req.sessionID);
  const username = req.session.username;

  const [rows, fields] = await db.query(
    "SELECT * FROM `Gardener` WHERE username=?",
    [username]
  );
  res.send({
    success: true,
    name: rows[0].name,
    employeeID: rows[0].employeeID,
    monthHours: rows[0].monthHours,
    phonenumber: rows[0].phonenumber,
    leaveDay: rows[0].leaveDay,
    garden: rows[0].garden,
  });
  }
);
```

Below is the code for Updating Gardner Details.

```
app.post("/api/addGardener", async (req, res) => {
  const username = req.session.username;
  const name = req.body.name;
  const phno = req.body.phonenumber;
  const employeeID = req.body.employeeID;
  const leaveDay = req.body.leaveDay;
  const garden = req.body.garden;

  const query =
    "UPDATE `Gardener` SET `name`=?,`phonenumber`=?,`employeeID`=?,`leaveDay`=?,`garden`=? WHERE username=?";
  try {
    db.query(
      query,
      [name, phno, employeeID, leaveDay, garden, username],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Details updated. Redirect to profile.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

Next up is a function **Locate Gardener:**

```
app.post("/api/locGardener", async (req, res) => {
  const username = req.session.username;
  const garden = req.body.garden;

  const query =
    "UPDATE `Gardener` SET `garden`=? WHERE username=?";
  try {
    db.query(
      query,
      [garden, username],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Location updated. Redirect to profile.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

Next up is the **Cut Requesting Code:**

```
app.post("/api/requestCut", async (req, res) => {
  const username = req.session.username;
  const garden = req.body.garden;

  const query =
    "INSERT INTO `CutRecs`(`garden`) VALUES (?)";
  try {
    db.query(
      query,
      [garden],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Cut Requested. Redirect to profile.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

There is also a **Garden Note Feature** as shown below:

```
app.post("/api/gardenNote", async (req, res) => {
  const username = req.session.username;
  const date = req.body.date;
  const garden = req.body.garden;
  const notes = req.body.notes;
  console.log(req.body);

  const query =
    "INSERT INTO `Notes` VALUES (?,?,?,?)";
  try {
    db.query(
      query,
      [date,username,garden,notes],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Note saved. Redirect to profile.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

Below are two codes. One for **Reading Notes** and the other for **Vendor Details.**

```
app.get("/api/getNotes", async (req, res) => {
  const query =
    "SELECT * from `Notes`";
  const [rows, fields] = await db.query(query, [req.session.username]);

  try {
    res.send({ success: true, info: rows });
  } catch (e) {
    console.log(e);
    res.send({ success: false });
  }
});

app.get("/api/getVendor", async (req, res) => {
  console.log("User cookie is", req.sessionID);
  const username = req.session.username;

  const [rows, fields] = await db.query(
    "SELECT * FROM `Vendor` WHERE username=?",
    [username]
  );
  res.send({
    success: true,
    vendorID: rows[0].vendorID,
    field: rows[0].field,
    dues: rows[0].dues,
  });
  }
);
```

Another necessary code is the **addDues** used by Vendors.

```
app.post("/api/addDues", async (req, res) => {
  const username = req.session.username;
  const date = req.body.date;
  const vendorID = req.body.vendorID;
  const equipID = req.body.equipID;
  const reason = req.body.reason;
  const dues = req.body.dues;

  const query =
    "INSERT INTO `VendorBills` VALUES (?,?,?,?,?,?)";
  try {
    db.query(
      query,
      [date,username,vendorID,equipID,reason,dues],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Note saved. Redirect to profile.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

Below is the code for **Repair Status Updates:**

```
app.post("/api/updateRepair", async (req, res) => {
  const username = req.session.username;
  const equipID = req.body.equipID;
  const repairStatus = req.body.repairStatus;

  const query =
    "UPDATE `Equipment` SET `repairStatus`=? WHERE equipID=?";
  try {
    db.query(
      query,
      [repairStatus, equipID],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Status updated. Redirect to profile.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

Finally, the **Manual Scheduling Code:**

```
app.post("/api/manualSchedule", async (req, res) => {
  const username = req.session.username;
  const day = req.body.day;
  const garden = req.body.garden;
  const number = req.body.number;

  const query =
    "UPDATE `WeekSchedule` SET `number`=?, `garden`=? WHERE day=?";
  try {
    db.query(
      query,
      [number, garden, day],
      (err, res) => {
        if (err) throw err;
        console.log("res is", res);
      }
    );
    res.send({
      success: true,
      message: "Day updated. Redirect to profile.",
    });
  } catch (e) {
    console.log(e);
    res.send({ success: false, message: "Something went wrong. Try again" });
  }
});
```

Along with these, there are some triggers and procedure I used which are shown below:

This is a trigger used to add details directly into tables on registration:

```
DELIMITER $$
CREATE TRIGGER add_landscape
AFTER INSERT
ON Users FOR EACH ROW
BEGIN
  IF new.RoleID = '4' THEN
    INSERT INTO Gardener VALUES(NULL, new.username, NULL, NULL, NULL, NULL, NULL, NULL );
  END IF;
END $$
DELIMITER ;
```

There is also a procedure to refresh weekly schedule table:

```
CREATE PROCEDURE refresh()
BEGIN
INSERT INTO WeekSchedule VALUES
('Monday',1,','),
('Tuesday',2,','),
('Wednesday',3,','),
('Thursday',4,','),
('Friday',5,','),
('Saturday',6,',');
END$$
```

This ends the list of important queries and codes used for the project.

Source code:

https://github.com/Vaishakh-SM/admin-portal

Drive link:

https://drive.google.com/drive/folders/1W8oCggDFO8GyjmZyjvlWzGIwcxjBjxZb?usp=sharing