CS F212 – DATABASE SYSTEMS

Gate Entry Management System [Documentation]

By –

Anish Ashish Kasegaonkar ***2020B3A70785P***

Abhinav Sahay ***2020B5A70788P***

Group No.: 117 **Project No.: 6**

Videos Link: https://drive.google.com/drive/u/0/folders/1ZfkyD2HnK0dRGgMvnD36u8UyZj2KHUaf

System Requirements Specifications:

- *Software Configuration*:
  - This Gate Entry Management System is developed using MySQL 8.0 by Oracle Corporation, as the backend to store the database
  - Node.js (1.0.0v) for making APIs and connecting MySQL (server-side scripting), Python for creating a CLI tool
  - Operating System: Linux, Solaris macOS, Windows
- *Hardware Configuration*:
  - Processor: Pentium(R) Dual-Core CPU and above
  - HardDisk: 40GB or more
  - RAM: 256 MB or more
- *Setup Configuration:*
  - Execute the gateSys.sql file on MySQLWorkbench

Basic Assumptions of the System:

- All drivers in the driver database are the only ones who can get entry into the gate.
- Only the vehicles registered under these drivers are allowed to enter the gates.
- Administrators do not have their own vehicles that need access to the gate.
- There is only 1 gate in our system (whereas in the real world, multiple gates can exit such as a back gate).
- Only vehicles are being tracked based on their VIN number; the system does not keep track of the entry/exit of people.
- All vehicles passing through the gate are in the log entries necessarily; there were no vehicles that passed through the gate before the table generation.

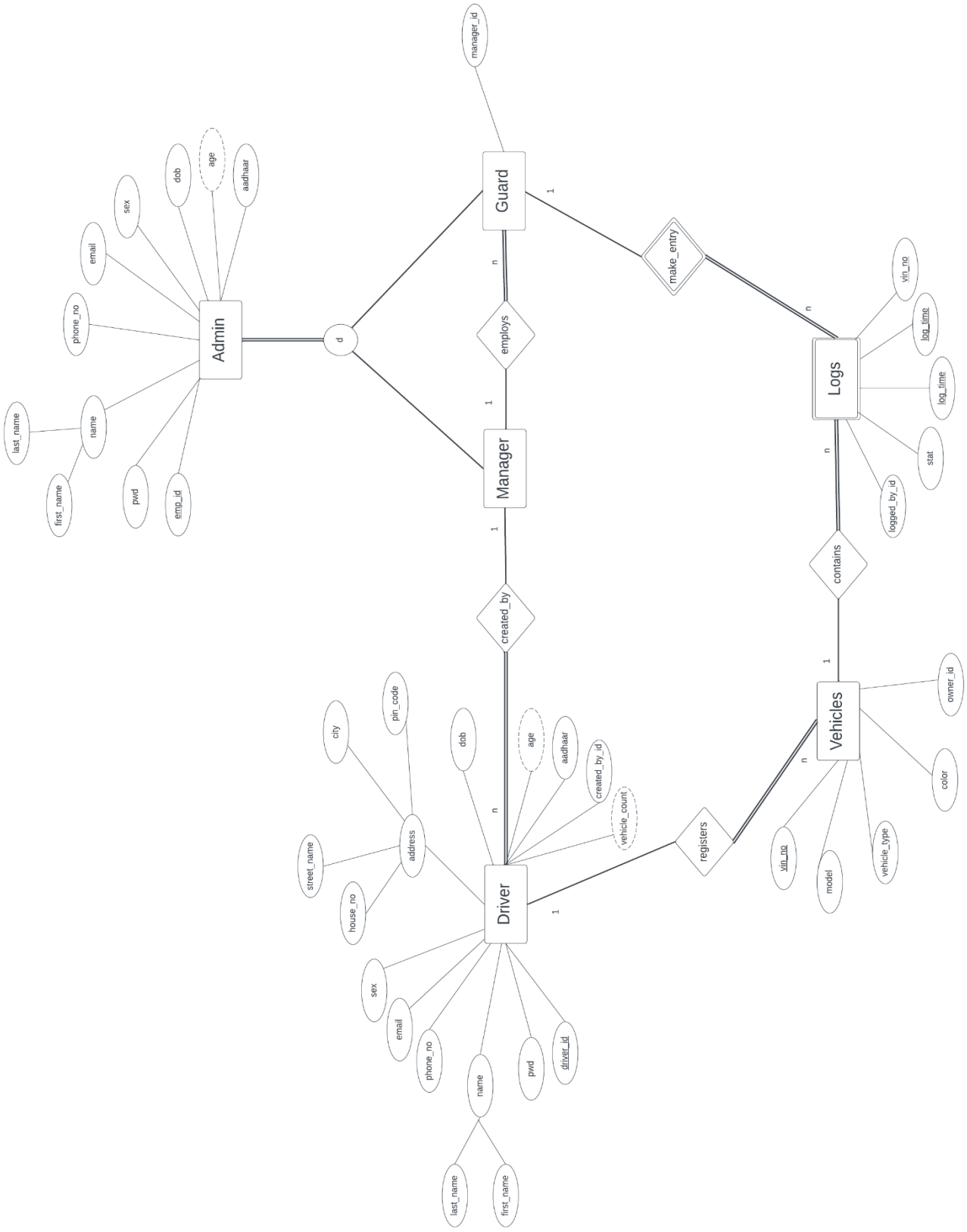
Entities:

- Five main entities - *drivers, vehicles, managers, guards and log entries. Log entries* is a weak entity which has an identifying relationship with *guards* and timestamp as a discriminator, and the others are strong entities.
- **Managers** are basically superusers, who can create driver profiles as well as enter into the logs. They hire guards and create guard profiles (more permissions than drivers). They can use the log data to generate reports and analytics on key metrics such as peak hours, vehicle types, gate utilization.
- **Guards** can view and edit log entries. They can also view the driver and vehicle tables but cannot edit them. Hence, they check if the vehicle is in the database, and if it is then they record their entry in the logs.
- **Drivers** can login to their profiles (created by the manager) and can edit certain fields such as phone number, address, vehicles owned, etc.
- **Vehicles** table consists of data related to each vehicle owned by a driver, such as VIN number, model, type, color, etc.
- **Log entries** table is essentially a logbook that consists of the entry and exit records of all vehicles whose entry is authorized by the guards. It consists of the vehicle's VIN number, the ID of the authorizing guard, a binary variable which specifies if the vehicle is entering or exiting, and a timestamp.

Relationships:

- **Drivers** can ***register*** their **vehicles**. Each vehicle is registered by exactly one owner (driver) and a driver can register multiple vehicles, hence the cardinality of this relationship is the binary 1:n type.
- **Log entries** ***contain*** information regarding **vehicles** that enter the gate. Each log entry contains exactly one vehicle and each vehicle can be a part of multiple log entries, hence the cardinality of this relationship is the binary 1:n type.
- **Guards** can ***make entries*** into the logbook by creating **log entries**. Each log entry is created by only one guard and a guard can create multiple such log entries, hence the cardinality of this relationship is also the binary 1:n type. This relationship is also an identifying relationship, in which log entries are dependent on guards.
- **Managers** can ***employ*** guards and create their profiles. Each guard can be employed by only one manager, and each manager can employ multiple guards, hence the cardinality of this relationship is also the binary 1:n type.
- **Managers** can ***create accounts*** for drivers. Each manager can create the profiles of multiple drivers, and each driver profile can be created by only one manager, hence the cardinality of this relationship is also the binary 1:n type.

The following is the **ER diagram** after removing the redundancies:



ER to Relational Model:

Step 1: *Mapping of strong entity types*
The driver, vehicle, manager and guard entities are strong entities. To map these in a relational model, a relation was created for each entity with all the simple attributes and the set of simple attributes forming each complex attribute. The key attribute of each entity is the primary key for the corresponding relation. For example, the vehicle entity is mapped to a vehicle relation with vin_no as the primary key.

Step 2: *Mapping of weak entity types*
The log entry is a weak entity. To map this in a relational model, a relation was created for it with all its simple attributes as well as foreign key attributes (primary key of guard relation). The primary key of this relation is the combination of the primary key of guard relation and the partial key (i.e. log_time and log_date).

Step 3: *Mapping of binary 1:1 relation types*
No such relations exist in the ER diagram.

Step 4: *Mapping of binary 1:N relation types*
All relations in the ER diagram are binary 1:N type of relations. As all the relationships have complete participation, it is possible to merge the relationship table within the entity table by including the primary key entity on the '1' side added as the foreign key entity on the 'N' side. For example, in the relation driver registers vehicles, a foreign key called "owner_id" referencing driver_id in the driver table was inserted into the vehicles table.

Step 5: *Mapping of binary M:N relation types*
No such relations exist in the ER diagram.

Step 6: *Mapping of multivalued attributes*
No such attributes exist in the ER diagram.

Step 7: *Mapping of N-ary relation types*
No such relations exist in the ER diagram.

Step 8: *Mapping specialization or generalization*
Administrators have been specialized into two disjoint subclasses, managers and guards. The subclass guard has a few more attributes, such as the ID of the manager who had employed them, and their joining date. Hence, a relation was created for each subclass, since the subclasses are total (every entity in the superclass belongs to at least one of the subclasses).

List of Functional Dependencies

- Driver relation:
    - *{driver_id} -> {pwd, first_name, last_name, phone_no, email, sex, house_no, street_name, city, pincode, dob, aadhar, created_by_id}*
    - *{pincode} -> {city}*
- Manager relation:
    - *{emp_id} -> {pwd, first_name, last_name, phone_no, email, sex, dob, aadhar}*
- Guard relation:
    - *{emp_id} -> {pwd, first_name, last_name, phone_no, email, sex, dob, aadhar, manager_id}*
- Logs relation:
    - *{vin_no, log_time, log_date} -> {stat, logged_by_id}*
- Vehicle relation:
    - *{vin_no} -> {model, vehicle_type, color, owner_id}*
    - *{model} -> {vehicle_type}*

Normalization

1NF:

- There are only single valued attributes.
- Attribute domain does not change for any attribute.
- There is a unique name for every attribute.
- The order in which data is stored does not matter.

Since all of the above conditions satisfy for all the generated relations, the database is in 1NF.

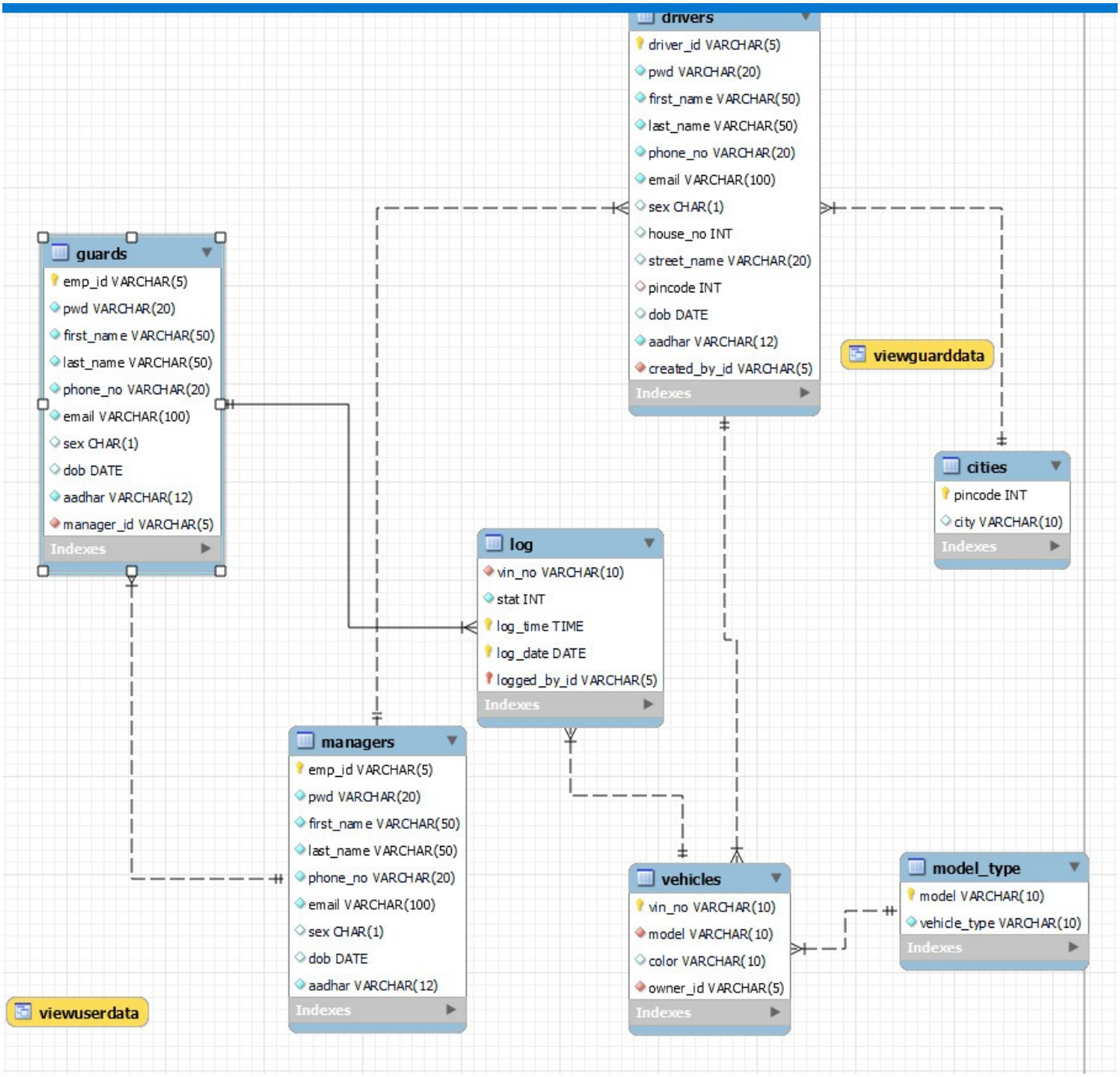
2NF:
A 1NF table is in 2NF form if and only if all of its non-prime attributes are functionally dependent on the whole of every candidate key, i.e. there doesn't exist any partial dependency. Since the previous condition holds for all the generated relations, the database is already in 2NF.

3NF:
A relation already in 2NF is in 3NF if there is no transitive dependency for non-prime attributes. However since all attributes are functionally dependent on the primary key, another way of stating the previous condition is that no non-prime attribute should determine another non-prime attribute. The relations in the tables driver (*{pincode} -> {city}*) and vehicle (*{model} -> {vehicle_type}*) are transitive dependencies, and to resolve this a new relation was created for each of these dependencies. Hence the corrected database with two new relations is now in 3NF.

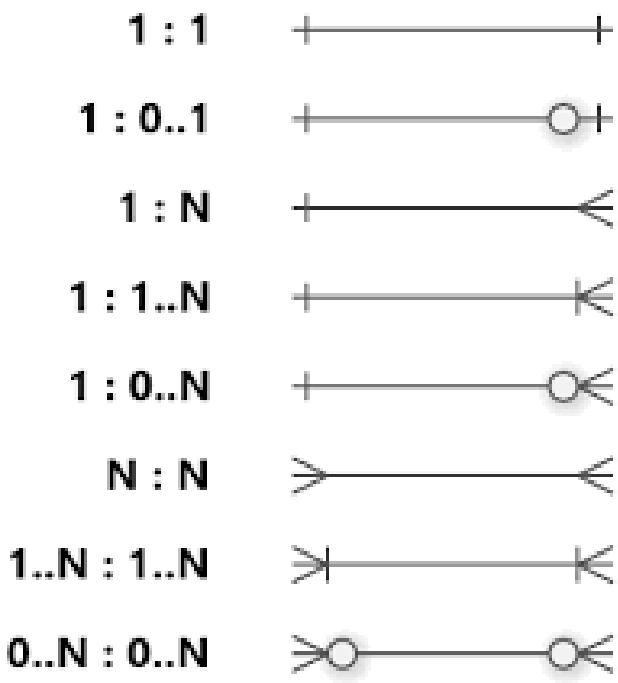
Foreign Keys and Candidate Keys

- Guards table contains a foreign key referencing the managers table (ID of the manager that they were employed by).
- Drivers table contains a foreign key referencing the managers table (ID of the manager who created their profile), as well as a foreign key referencing the cities table (to display the corresponding city for each pincode in driver view).
- Vehicles table contains a foreign key referencing the drivers table (ID of the driver owning the vehicle), as well as a foreign key referencing the model_type table (to display the corresponding vehicle type for each vehicle in vehicle view).
- Logs entries table contains a foreign key referencing the vehicles table (VIN number of the corresponding vehicle), as well as a foreign key referencing the guards table (ID of the guard who made the entry).
- All candidate keys are primary keys of the corresponding tables (mentioned in the final schema below).

The following **schema** lays out the logical structure for the database and helps translate the data model into specific tables, columns, keys and interrelations.



The different tables were merged with the relationship tables from the ER diagram to reduce redundancy wherever possible. The schema design follows the crow's foot notation.



The following was the resulting schema design:

1. *driver(driver_id, pwd, first_name, last_name, phone_no, email, sex, house_no, street_name, pincode, dob, aadhar, created_by_id)*
2. *managers(emp_id, pwd, first_name, last_name, phone_no, email, sex, dob, aadhar)*
3. *guards(emp_id, pwd, first_name, last_name, phone_no, email, sex, dob, aadhar, manager_id)*
4. *vehicles(vin_no, model, color, owner_id)*
5. *logs(vin_no, log_time, log_date, stat, logged_by_id)*
6. *cities(pincode, city)*
7. *model_type(model, vehicle_type)*

Screenshots of Program Runs

1. Create profile for users accessing the gate

Procedure:

```
DROP PROCEDURE IF EXISTS create_user;
DELIMITER $$
CREATE PROCEDURE create_user(id varchar(5), fname varchar(50), lname varchar(50), phone varchar(20), mail varchar(50), sex varchar(1), house int, road varchar(20), new_city varchar(10), new_pincode int, birth date, aadhar_no varchar(12), man_id varchar(5))
    MODIFIES SQL DATA
BEGIN
    IF (new_pincode IN (SELECT pincode from cities) AND new_city IN (SELECT city FROM cities WHERE new_pincode = cities.pincode)) THEN
        INSERT INTO drivers (driver_id, first_name, last_name, phone_no, email, sex, house_no, street_name, pincode, dob, aadhar, created_by_id) VALUES (id, fname, lname, phone, mail, sex, house, road, new_pincode, birth, aadhar_no, man_id);
    ELSEIF (new_pincode IN (SELECT pincode from cities) AND new_city NOT IN (SELECT city FROM cities WHERE new_pincode = cities.pincode)) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT="INCORRECT CITY!";
    ELSE
        INSERT INTO cities(pincode, city) values(new_pincode, new_city);
        INSERT INTO drivers (driver_id, first_name, last_name, phone_no, email, sex, house_no, street_name, pincode, dob, aadhar, created_by_id) VALUES (id, fname, lname, phone, mail, sex, house, road, new_pincode, birth, aadhar_no, man_id);
    END IF;
END$$
DELIMITER ;
```

Previous data in drivers table:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

driver_id	pwd	first_name	last_name	phone_no	email	sex	house_no	street_name	pincode	dob	aadhar	created_by_id
D001	driver123	Alex	Chung	9876543217	alex.chung@example.com	M	15	Main Street	110001	1995-11-23	567890123456	M001
D002	driver456	Emma	Lee	9876543218	emma.lee@example.com	F	25	Second Street	400001	1997-03-14	789012345678	M001
D003	driver789	Tom	Park	9876543219	tom.park@example.com	M	35	Third Street	500001	1992-05-06	901234567890	M002
D004	driverabc	Olivia	Kim	9876543220	olivia.kim@example.com	F	45	Fourth Street	110002	1996-08-29	123456789012	M003
D005	driver005	Jacob	Wu	8125077892	jacob.wu@example.com	M	20	Fifth Street	400011	1998-01-15	238479102938	M004
D006	driver006	Ella	Zhang	8125077893	ella.zhang@example.com	F	30	Sixth Street	500001	1990-12-02	749203849012	M004
D007	driver007	Noah	Chang	8765432102	noah.chang@example.com	M	40	Seventh Street	110001	1987-09-22	039485739482	M005
D008	driver008	Sophia	Choi	7712495099	sophia.choi@example.com	F	50	Eighth Street	400001	1986-11-17	748291048593	M005
D009	driver009	William	Kang	9993914808	william.kang@example.com	M	18	Ninth Street	500012	2004-02-20	897491230097	M006
D010	driver010	Grace	Kim	9297004139	grace.kim@example.com	F	28	Tenth Street	110001	1995-06-28	146723804390	M006
D011	driver011	Ethan	Park	9880231477	ethan.park@example.com	M	38	Eleventh Street	400001	1984-03-31	698321427098	M007
D012	driver012	Chloe	Lee	9993918808	chloe.lee@example.com	F	48	Twelfth Street	500001	1974-12-19	379499182498	M007
D013	driver013	Daniel	Chen	9889078724	daniel.chen@example.com	M	22	Thirteenth Str...	110001	1999-08-05	567898123456	M008
D014	driver014	Lily	Li	8765412102	lily.li@example.com	F	32	Fourteenth St...	400001	1989-04-25	789612345678	M008
D015	driver015	Luke	Yang	8125077894	luke.yang@example.com	M	42	Fifteenth Street	600001	1980-07-14	909234567890	M009
D016	driver016	Ava	Jung	8125077895	ava.jung@example.com	F	52	Sixteenth Street	110001	1970-02-18	123456789092	M009

New data in drivers table:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

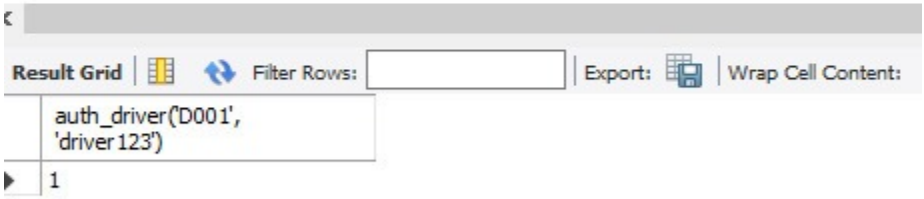
driver_id	pwd	first_name	last_name	phone_no	email	sex	house_no	street_name	pincode	dob	aadhar	created_by_id
D001	driver123	Alex	Chung	9876543217	alex.chung@example.com	M	15	Main Street	110001	1995-11-23	567890123456	M001
D002	driver456	Emma	Lee	9876543218	emma.lee@example.com	F	25	Second Street	400001	1997-03-14	789012345678	M001
D003	driver789	Tom	Park	9876543219	tom.park@example.com	M	35	Third Street	500001	1992-05-06	901234567890	M002
D004	driverabc	Olivia	Kim	9876543220	olivia.kim@example.com	F	45	Fourth Street	110002	1996-08-29	123456789012	M003
D005	driver005	Jacob	Wu	8125077892	jacob.wu@example.com	M	20	Fifth Street	400011	1998-01-15	238479102938	M004
D006	driver006	Ella	Zhang	8125077893	ella.zhang@example.com	F	30	Sixth Street	500001	1990-12-02	749203849012	M004
D007	driver007	Noah	Chang	8765432102	noah.chang@example.com	M	40	Seventh Street	110001	1987-09-22	039485739482	M005
D008	driver008	Sophia	Choi	7712495099	sophia.choi@example.com	F	50	Eighth Street	400001	1986-11-17	748291048593	M005
D009	driver009	William	Kang	9993914808	william.kang@example.com	M	18	Ninth Street	500012	2004-02-20	897491230097	M006
D010	driver010	Grace	Kim	9297004139	grace.kim@example.com	F	28	Tenth Street	110001	1995-06-28	146723804390	M006
D011	driver011	Ethan	Park	9880231477	ethan.park@example.com	M	38	Eleventh Street	400001	1984-03-31	698321427098	M007
D012	driver012	Chloe	Lee	9993918808	chloe.lee@example.com	F	48	Twelfth Street	500001	1974-12-19	379499182498	M007
D013	driver013	Daniel	Chen	9889078724	daniel.chen@example.com	M	22	Thirteenth Str...	110001	1999-08-05	567898123456	M008
D014	driver014	Lily	Li	8765412102	lily.li@example.com	F	32	Fourteenth St...	400001	1989-04-25	789612345678	M008
D015	driver015	Luke	Yang	8125077894	luke.yang@example.com	M	42	Fifteenth Street	600001	1980-07-14	909234567890	M009
D016	driver016	Ava	Jung	8125077895	ava.jung@example.com	F	52	Sixteenth Street	110001	1970-02-18	123456789092	M009
D111	123	Afewex	Brad	9877843217	brad@example.com	M	25	Main	110921	1995-11-23	567890193456	M001

2. Sign-in authentication to verify if provided credentials by users are correct

Procedure:

```
-- PROCEDURES --
DROP FUNCTION IF EXISTS auth_driver;
DELIMITER $$
CREATE FUNCTION auth_driver(id VARCHAR(5), pass VARCHAR(20))
    RETURNS INT
    READS SQL DATA
BEGIN
    DECLARE auth_value INT;
    SELECT COUNT(*) INTO auth_value FROM drivers WHERE driver_id = id AND pwd = pass;
    RETURN auth_value;
END$$
DELIMITER ;
```

When the username/password is correct:



When the username/password is incorrect:

```
1 •  SELECT auth_driver('D001', 'driver12');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

auth_driver('D001', 'driver12')
0

3. Password reset to update the stored password against username (Security question is the user's Aadhar number)

Procedure:

```
DROP FUNCTION IF EXISTS reset_password_drivers;
DELIMITER $$
CREATE FUNCTION reset_password_drivers(id varchar(5), sec_aadhar VARCHAR(12), new_pass VARCHAR(20))
    RETURNS INT
    DETERMINISTIC
BEGIN
    declare auth INT;
    IF (sec_aadhar = (SELECT aadhar from drivers where driver_id = id)) THEN
        UPDATE drivers SET drivers.pwd = new_pass where drivers.aadhar = sec_aadhar;
        SET auth = 1;
        return auth;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "FAILED TO UPDATE PASSWORD! PLEASE CHECK DETAILS!";
        SET auth = -1;
        return auth;
    END IF;
END$$
DELIMITER ;
```

Old password:

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

driver_id	pwd	first_name	last_name	phone_no	email	sex	house_no	street_name	pincode	dob	aadhar	created_by_id
D001	driver123	Alex	Chung	9876543217	alex.chung@example.com	M	15	Main Street	110001	1995-11-23	567890123456	M001

Success in resetting password (as Aadhar was entered correctly):

```
1 • SELECT reset_password_drivers('D001', '567890123456', 'newpass');
2 • select * from drivers limit 1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

reset_password_drivers('D001', '567890123456', 'newpass')
1

New password:

driver_id	pwd	first_name	last_name	phone_no	email	sex	house_no	street_name	pincode	dob	aadhar	created_by_id
D001	newpass	Alex	Chung	9876543217	alex.chung@example.com	M	15	Main Street	110001	1995-11-23	567890123456	M001

4. User should be able to manage the profile, i.e. update phone number, etc

Procedure:

```
DROP PROCEDURE IF EXISTS update_user_phone;
DELIMITER $$
CREATE PROCEDURE update_user_phone(id varchar(5), new_number varchar(20))
    MODIFIES SQL DATA
BEGIN
    IF (id in (SELECT driver_id FROM drivers)) THEN
        UPDATE drivers SET drivers.phone_no = new_number where drivers.driver_id = id;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "FAILED TO UPDATE NUMBER!";
    END IF;
END$$
DELIMITER ;
```

Old phone number:

driver_id	pwd	first_name	last_name	phone_no	email	sex	house_no	street_name	pincode	dob	aadhar	created_by_id
D001	newpass	Alex	Chung	9876543217	alex.chung@example.com	M	15	Main Street	110001	1995-11-23	567890123456	M001

New phone number:

```
call update_user_phone('D001', '9993989303');
select * from drivers limit 1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

driver_id	pwd	first_name	last_name	phone_no	email	sex	house_no	street_name	pincode	dob	aadhar	created_by_id
D001	newpass	Alex	Chung	9993989303	alex.chung@example.com	M	15	Main Street	110001	1995-11-23	567890123456	M001

5. User can register the vehicles owned

Procedure:

```
DROP PROCEDURE IF EXISTS register_vehicle;
DELIMITER $$
CREATE PROCEDURE register_vehicle(id varchar(5), new_vin_no varchar(10), new_model varchar(10), new_type varchar(10), new_color varchar(10))
    MODIFIES SQL DATA
BEGIN
    IF (id IN (SELECT driver_id FROM drivers)) THEN
        IF (new_model IN (SELECT model FROM model_type) AND new_type IN (SELECT vehicle_type FROM model_type WHERE new_model = model_type.model)) THEN
            INSERT INTO vehicles(owner_id, vin_no, model, color) values (id, new_vin_no, new_model, new_color);
        ELSEIF (new_model IN (SELECT model FROM model_type) AND new_type NOT IN (SELECT vehicle_type FROM model_type WHERE new_model = model_type.model)) THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "INVALID MODEL TYPE!";
        ELSE
            INSERT INTO model_type(model, vehicle_type) values (new_model, new_type);
            INSERT INTO vehicles(owner_id, vin_no, model, color) values (id, new_vin_no, new_model, new_color);
        END IF;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "FAILED TO REGISTER VEHICLE!";
    END IF;
END$$
DELIMITER ;
```

List of previously owned vehicles:

vin_no	model	color	owner_id
VIN001	M1	Red	D001
VIN002	M1	Blue	D002
VIN003	M2	Pink	D002
VIN004	M3	Silver	D003
VIN005	M4	Black	D004
VIN006	M2	White	D005
VIN007	M5	Green	D006
VIN008	M1	Yellow	D007
VIN009	M2	Orange	D007
VIN010	M3	Red	D008
VIN011	M4	Blue	D009
VIN012	M5	Pink	D009
VIN013	M1	Silver	D009
VIN014	M2	Black	D011
VIN015	M3	White	D012
VIN016	M4	Green	D013
VIN017	M5	Yellow	D014
VIN018	M1	Orange	D015
VIN019	M2	Red	D016

New list of owned vehicles:

```
1    call register_vehicle('D001', 'VIN020', 'M10', 'Auto', 'Black');
2 •  select * from vehicles;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vin_no	model	color	owner_id
VIN001	M1	Red	D001
VIN002	M1	Blue	D002
VIN003	M2	Pink	D002
VIN004	M3	Silver	D003
VIN005	M4	Black	D004
VIN006	M2	White	D005
VIN007	M5	Green	D006
VIN008	M1	Yellow	D007
VIN009	M2	Orange	D007
VIN010	M3	Red	D008
VIN011	M4	Blue	D009
VIN012	M5	Pink	D009
VIN013	M1	Silver	D009
VIN014	M2	Black	D011
VIN015	M3	White	D012
VIN016	M4	Green	D013
VIN017	M5	Yellow	D014
VIN018	M1	Orange	D015
VIN019	M2	Red	D016
VIN020	M10	Black	D001

6. Query to search for vehicle record based on filters such as pincode of owner's address

Procedure:

```
DROP PROCEDURE IF EXISTS search_vehicle_by_pin;
DELIMITER $$
CREATE PROCEDURE search_vehicle_by_pin(pin int)
    READS SQL DATA
BEGIN
    IF (pin IN (SELECT pincode FROM drivers)) THEN
        SELECT a.vin_no, a.model, b.vehicle_type, a.color, a.owner_id FROM vehicles a INNER JOIN model_type b ON a.model = b.model AND a.owner_id IN (SELECT driver_id FROM drivers WHERE drivers.pincode = pin);
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "NO VEHICLE FOUND!";
    END IF;
END$$
DELIMITER ;
```

Result of query:

```
call search_vehicle_by_pin('110001');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vin_no	model	vehicle_type	color	owner_id
VIN001	M1	Car	Red	D001
VIN020	M10	Auto	Black	D001
VIN008	M1	Car	Yellow	D007
VIN009	M2	SUV	Orange	D007
VIN016	M4	Truck	Green	D013
VIN019	M2	SUV	Red	D016

7. Gate Authority should be able to record entry/ exit times

Procedure:

```
DROP PROCEDURE IF EXISTS record_exit_time;
DELIMITER $$
CREATE PROCEDURE record_exit_time(guard_id varchar(5), vehicle_no varchar(10))
    MODIFIES SQL DATA
BEGIN
    IF (vehicle_no IN (SELECT vin_no FROM vehicles)) THEN
        IF ((SELECT  d1.stat FROM log AS d1 LEFT OUTER JOIN log AS d2 ON d1.log_date < d2.log_date OR d1.log_date = d2.log_date and d1.log_time < d2.log_time WHERE (d2.vin_no IS NULL)) = 1) THEN
            INSERT INTO log(vin_no, stat, log_time, log_date, logged_by_id) VALUES (vehicle_no, 0, current_time(), current_date(), guard_id);
        ELSE
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "VEHICLE ALREADY OUTSIDE LOT!";
        END IF;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "VEHICLE NOT REGISTERED!";
    END IF;

END$$
DELIMITER ;
```

Previous logs (with VIN016 having 1, i.e. inside):

VIN003	1	13:00:00	2008-01-03	G010
VIN014	0	13:00:00	2014-06-03	G010
VIN005	0	16:30:00	2021-09-22	G010
VIN003	0	16:00:00	2010-09-29	G011
VIN012	0	16:00:00	2015-11-19	G011
VIN003	1	18:30:00	2012-11-14	G012
VIN015	0	18:30:00	2016-03-29	G012
VIN004	0	15:30:00	2007-06-17	G013
VIN016	1	15:30:00	2017-05-10	G013
VIN004	1	18:00:00	2007-12-03	G014
VIN007	1	18:00:00	2018-01-05	G014
VIN004	0	19:00:00	2008-08-22	G015
VIN019	1	19:00:00	2019-03-22	G015
VIN004	1	20:30:00	2009-05-06	G016
VIN009	1	20:30:00	2020-06-09	G016
VIN004	0	10:00:00	2011-07-12	G017
VIN011	1	10:00:00	2021-04-17	G017
VIN017	1	08:45:00	2023-02-19	G018
VIN004	1	12:00:00	2012-02-28	G018
VIN018	1	12:00:00	2022-01-01	G018
VIN013	1	15:30:00	2022-08-10	G018

Recorded exit time:

```
select * from log;
call record_exit_time('G001', 'VIN016');
```

Result Grid | Filter Rows: | Export: | Wrap

vin_no	stat	log_time	log_date	logged_by_id
VIN003	1	13:00:00	2008-01-03	G010
VIN014	0	13:00:00	2014-06-03	G010
VIN005	0	16:30:00	2021-09-22	G010
VIN003	0	16:00:00	2010-09-29	G011
VIN012	0	16:00:00	2015-11-19	G011
VIN003	1	18:30:00	2012-11-14	G012
VIN015	0	18:30:00	2016-03-29	G012
VIN004	0	15:30:00	2007-06-17	G013
VIN016	1	15:30:00	2017-05-10	G013
VIN004	1	18:00:00	2007-12-03	G014
VIN007	1	18:00:00	2018-01-05	G014
VIN004	0	19:00:00	2008-08-22	G015
VIN019	1	19:00:00	2019-03-22	G015
VIN004	1	20:30:00	2009-05-06	G016
VIN009	1	20:30:00	2020-06-09	G016
VIN004	0	10:00:00	2011-07-12	G017
VIN011	1	10:00:00	2021-04-17	G017
VIN017	1	08:45:00	2023-02-19	G018
VIN004	1	12:00:00	2012-02-28	G018
VIN018	1	12:00:00	2022-01-01	G018
VIN013	1	15:30:00	2022-08-10	G018

8. Recorded entry-exit time can be retrieved by the user

Procedure:

```
DROP PROCEDURE IF EXISTS obtain_logs;
DELIMITER $$
CREATE PROCEDURE obtain_logs(user_id varchar(5))
    READS SQL DATA
BEGIN
    IF (user_id IN (SELECT driver_id FROM drivers)) THEN
        SELECT a.vin_no, a.stat, a.log_time, a.log_date from log a INNER JOIN vehicles b ON a.vin_no = b.vin_no and b.owner_id = user_id;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "UNABLE TO RETRIEVE LOGS";
    END IF;
END$$
DELIMITER  ;
```

Obtained logs:

```
call obtain_logs('D001');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vin_no	stat	log_time	log_date
VIN001	0	20:30:00	2003-11-03
VIN001	0	11:00:00	2008-08-12
VIN001	1	11:51:00	2009-12-09
VIN001	1	15:00:00	2011-09-02
VIN001	1	20:51:00	2004-02-13
VIN001	0	10:30:00	2013-07-18
VIN001	1	12:15:00	2016-09-19
VIN001	0	16:30:00	2018-02-23
VIN001	1	10:30:00	2019-10-02

9. **Reports can be generated on key metrics such as peak hours, vehicle types, gate utilization**

Procedure:

```
DROP PROCEDURE IF EXISTS obtain_peak_hour;
DELIMITER $$
CREATE PROCEDURE obtain_peak_hour()
    READS SQL DATA
BEGIN
    select date_format( log_time, '%H' ) as `hour`, count('hour') from log group by date_format( log_time, '%H' ) order by count(*) desc limit 1;
END$$
DELIMITER ;
```

Obtained peak hours:

```
1    call obtain_peak_hour();
```

esult Grid | Filter Rows: | Export:

hour	count('hour')
10	9

Similar functions have been created for registered models, and number of vehicles inside and outside the lot (and has been explained in detail below).

Procedures, functions and views

- **auth_driver() -** Returns 1 if the username and password entered by the user corresponds to some driver in the driver table.
- **auth_manager() -** Returns 1 if the username and password entered by the user corresponds to some manager in the manager table.
- **auth_guard() -** Returns 1 if the username and password entered by the user corresponds to some guard in the guard table.
- **reset_password_drivers() -** Resets the driver's password if they enter their Aadhar number correctly.
- **reset_password_manager() -** Resets the manager's password if they enter their Aadhar number correctly.
- **reset_password_guard() -** Resets the guard's password if they enter their Aadhar number correctly.
- **register_vehicle() -** Registers a vehicle under a driver. If the vehicle model type doesn't exist in the model_type table, it first makes the appropriate changes in that table before appending to the vehicle table. If the vehicle type is incorrectly entered corresponding to the entry in the model_type table, then an error message is displayed.
- **deregister_vehicle() -** Deregisters a vehicle that was previously registered under a driver. Has checks to ensure that the vehicle was previously in the table.
- **search_vin_no() -** Searches for a vehicle on the basis of its VIN number.
- **search_owner_name() -** Searches for all vehicles registered under a particular owner.
- **search_vehicle_by_pin() -** Searches for all vehicles registered by owner's who live at addresses with a particular pincode.
- **record_entry_time() -** A functionality to help guards record entry of a vehicle. If the vehicle exists already in the vehicle database, then a log entry is created. If the vehicle does not exist in the vehicle database or if the vehicle is already inside the gate, then it is not allowed entry.
- **record_exit_time() -** A functionality to help guards record exit of a vehicle. If the vehicle exists already in the vehicle database, then a log entry is

created. If the vehicle does not exist in the vehicle database or if the vehicle is already outside the gate, then it is not allowed exit.

- **obtain_logs() -** Obtains all the log entries using an inner join between the log and vehicles tables.
- **update_user_phone() -** A functionality for a particular driver to go to their profiles and update their phone numbers.
- **update_user_email() -** A functionality for a particular driver to go to their profiles and update their email IDs.
- **update_user_address() -** A functionality for a particular driver to go to their profiles and update their address (which includes house number, street name, city and pincode). Additional checks to ensure that the pincode corresponds to the correct city in the cities table.
- **obtain_user_data() -** A functionality that returns the driver's profile data to be displayed once they login.
- **obtain_peak_hour() -** A report that can be generated by the managers, that displays the peak hours of gate utilization. Basically, it displays all the hours of the day in which at least one vehicle has passed through the gate, with a count of the number of vehicles who have passed either inside or outside the gate, in a descending order of this count.
- **obtain_registered_models() -** A report that can be generated by the managers, that displays the different types of vehicles that pass through the gate and its count. This data again, is sorted in descending order on the basis of this count, and can be used for further analytics.
- **obtain_vehicles_in() -** A report that can be generated by the managers, that displays the count of all unique vehicles that are inside the gate.
- **obtain_vehicles_out() -** A report that can be generated by the managers, that displays the count of all unique vehicles who had entered the gate at some point of time, but are now outside the gate.
- **create_user() -** A functionality that is available only to the managers, so that they can register new drivers in the driver table.
- **create_guard() -** A functionality that is available only to the managers, so that they can register new guards in the guards table.
- **delete_user() -** A functionality that is available only to the managers, so that they can remove drivers existing in the driver table. Additional checks to ensure that the driver exists previously in the driver table.

- **delete_guard() -** A functionality that is available only to the managers, so that they can remove guards existing in the guards table. Additional checks to ensure that the guard exists previously in the guards table.
- **viewGuardData -** A view generated to display the details of guards to the managers, along with derived attributes mentioned in the ER such as age of each guard in the guard table.
- **viewUserData -** A view generated to display the details of drivers to the managers, along with derived attributes mentioned in the ER such as age and count of vehicles owned by each driver in the driver table.

Concurrency and consistency

The database achieves concurrency on a basic level by ensuring that all queries are atomic in nature, and hence act in a sense just like transactions, i.e. are either executed fully or not at all. The queries are executed in a serialized manner hence removing the problem of inconsistency. All internal updates, like when a driver changes their phone number or email ID, occur simultaneously and hence help in maintaining a consistent state for the complete database. Internal Update/ Delete triggers like Cascading, Restricting and No Action were used to ensure the consistency of data as and where required. Hence Update and Delete Anomalies were taken care of using cases like:

- Removal of a driver from the drivers table (which ensures that their vehicles are also deleted from the database). If the driver was the only driver who lived in a particular pincode, then the appropriate entry would also be removed from the cities table.
- Removal of a driver's vehicle from the vehicles table (which ensures that the corresponding data from the model_type table is also deleted in the case that the deleted vehicle was the only vehicle of the particular model).
- Removal of a manager would lead to the removal of all guards that were reporting under the manager.

Future improvements

- More complex transactions can be added in the future, complete with commits and rollbacks.
- The superuser should have a choice to reallocate the guards to a different manager if required.
- Streamlining of the gate entry management process, in terms of efficient allocation of guards at gates on the basis of reports (in a multi-gate system) to minimize costs.
- Use of Update/Delete/Insert Triggers can be made to ensure preconditions and reinforce consistency.
- More features such as entry for visitors, where a notification would go to a particular driver and they would get a choice to let the visitor in or not.
- User-friendly UI (frontend) so that users can use the application easily, integrated with back-end code of the database. We had attempted to build this, and a full back-end code (in NodeJS) tested on Postman has been included in the submission.