ÉCOLE SUPÉRIEURE EN SCIENCES ET TECHNOLOGIES
DE L'INFORMATIQUE ET DU NUMÉRIQUE

First year of higher education

1CS

BDD Projcet

EasyPark - Private Parking Website

Created by:

Batiche Chourouq

Boufroura Rania

Kiouaz Selssabila

Supervised by:

Dr. Lekehali Somia

Ms. Ladlani Randa

Mr. Hammoum Yanis

Submission date: January 27, 2024
Academic year: 2023-2024

Thanks

First and foremost, we express our gratitude to Allah the Almighty, who granted us the courage and patience needed to complete this work.

Our heartfelt thanks go to our supervisors, **Dr.Lekehali Somia** and **Ms.Ladlani Randa**, **Mr.Hammoum Yanis** for their competent assistance, patience, and constant support. Their critical insights proved invaluable in structuring our work and enhancing the quality of different sections of the project.

We also extend our gratitude to our dear parents for their unwavering support and patience, which have been the pillars enabling us to reach the stage where we are today.

Our thanks also go to all our relatives, friends, and teachers at ESTIN, whose constant encouragement has been a source of inspiration throughout the realization of this project.

Lastly, we want to express our appreciation to all individuals who contributed, directly or indirectly, to the realization of this work. Your contribution has been invaluable, and we are grateful to have shared this journey with you.

Contents

1 Inroduction **2**

1.1 Scenario 3

1.2 Purpose of the Report 3

1.2.1 Objectives in Real Life 3

1.2.2 Importance 3

1.2.3 Analysis of Requirements 4

1.2.4 Conception Phase Using Entity-Relationship Diagram (ERD) . 4

1.2.5 Data Dictionary 4

1.2.6 Integrity Constraints 4

1.2.7 SQL Queries and Examples 4

1.2.8 Tools Used 5

2 Requirement Analysis **6**

2.1 Requirement Analysis 7

2.1.1 Functional Requirements 7

2.1.2 Non-functional Requirements 8

2.1.3 Requirements Gathering 8

3 Conception of the project **10**

3.1 Conception Phase Using Entity-Relationship Diagram (ERD) 11

3.1.1 Entity-Relationship Diagram (ERD) 11

3.1.2 Entities 12

3.1.3 Relationships and Cardinalities 12

3.1.4 Rationale 13

3.2 Data Dictionary 14

4 Development of the project **17**

4.1 Integrity Constraints 18

4.1.1 Users Table 18

4.1.2 Admin Table 18

4.1.3 ClientHistory Table 18

4.1.4 Spots Table 18

4.1.5 Plans Table 18

4.1.6 ActivityLog Table 19

4.1.7 Subscription Table 19

4.1.8 Notification Table 19

4.1.9 Reservation Table 19

4.2 SQL Queries and Examples 19

4.2.1 Inserting a User 19

4.2.2 Search Users by Name 20

4.2.3 Update User Profile 20

4.2.4 Get Admin Profile 20

4.2.5 Create Reservation 20

4.2.6 Search Reservations by Date 20

4.2.7 Update Reservation Status 20

4.2.8 Update Plan 20

4.2.9 Get Plan by ID 20

4.2.10 Delete User 21

4.2.11 Create Notificationr 21

4.2.12 Get All Notifications 21

4.2.13 Retrieving a User by Email 21

4.2.14 Inserting a Client History Record 21

4.2.15 Retrieving Client History 21

4.3 Query Descriptions 21

4.3.1 Inserting a User 21

4.3.2 Search Users by Name 22

4.3.3 Update User Profile 22

4.3.4 Get Admin Profile 22

4.3.5 Create Reservation 22

4.3.6 Search Reservations by Date . . . 22
4.3.7 Update Reservation Status . . . 23
4.3.8 Update Plan . . . 23
4.3.9 Get Plan by ID . . . 23
4.3.10 Delete User . . . 23
4.3.11 Create Notification . . . 23
4.3.12 Get All Notifications . . . 24
4.3.13 Retrieving a User by Email . . . 24
4.3.14 Inserting a Client History Record . . . 24
4.3.15 Retrieving Client History . . . 24
4.3.16 User Registration fonction . . . 25
4.3.17 User Login fonction . . . 25
4.3.18 Inserting a Client History Record . . . 26
4.3.19 Client History Retrieval . . . 26
4.4 Tools Used . . . 27
4.4.1 Communication Tools . . . 27
4.4.2 Usage of the VS IDE (Visual Studio) . . . 28
4.4.3 Design Tool . . . 29
4.4.4 Front-end Tools . . . 29
4.4.5 Backend Tools . . . 31
4.5 Presentation of Graphic Interfaces . . . 33
4.5.1 Presentation of Graphic Interfaces . . . 33

5 Conclusion **42**
5.1 Conclusion . . . 43

List of Figures

3.1 Entity-Relationship Diagram for the Private Parking Management System 11

4.1 User Registration fonction 25
4.2 User Login fonction 25
4.3 Inserting a Client History Record 26
4.4 Client History Retrieval 26
4.5 Logo de Google meet 28
4.6 Logo de Messenger 28
4.7 Logo de Github 28
4.8 Logo de Gmail 28
4.9 figma. 29
4.10 Logo de tailwind 30
4.11 Logo de react.js 30
4.12 Logo de vite 30
4.13 Access the database using SQL shell. 32
4.14 Part of the database schema 32
4.15 Interface to create a server 33
4.16 Postman interface to test API for creating a user 33
4.17 Homepage - Welcome 34
4.18 Homepage - Services 34
4.19 Homepage - About Us 35
4.20 Homepage - contact 35
4.21 Authentification-login 36
4.22 Authentification-Sign-Up 36
4.23 Client Interface - Home 37
4.24 Client Interface - Reservation Service 37

4.25 Client Interface - History of Past Reservations 38
4.26 Client Interface - Select a Subscription 38
4.27 Admin-Dashboard(Spots) 39
4.28 Admin-Dashboard(User management) 39
4.29 Admin-Profile 40
4.30 Admin-Edit profile 40
4.31 Admin-notification 41

List of Tables

3.1 Admin Table Data Dictionary 14

3.2 Client Table Data Dictionary 14

3.3 ActivityLog Table Data Dictionary 14

3.4 ClientHistory Table Data Dictionary 15

3.5 Reservation Table Data Dictionary 15

3.6 Subscription Table Data Dictionary 15

3.7 Spot Table Data Dictionary 16

3.8 Plan Table Data Dictionary 16

3.9 Manage Relationship Table Data Dictionary 16

Chapter 1

Inroduction

1.1 Scenario

The scenario chosen for this database application is a private parking management system. In our day-to-day lives, parking spaces are often a valuable commodity, and efficiently managing them can be a challenging task. This database application aims to streamline the process of reserving parking spaces, handling user subscriptions, and providing administrative tools for effective management.

Parking scenarios involve dynamic interactions, with users seeking available spots, making reservations, and subscribing to long-term plans. The database application will model these real-life entities, ensuring a seamless and organized parking experience for users.

1.2 Purpose of the Report

This report serves a dual purpose: to document and to analyze the private parking management system. Through a detailed exploration of the system's development, implementation, and functionalities, the report aims to provide a comprehensive understanding of the project. The key objectives include:

1.2.1 Objectives in Real Life

- Efficient Parking Space Utilization: Optimize the utilization of available parking spaces, ensuring that each spot is used effectively.
- User Convenience: Enhance the overall experience for users by providing easy access to parking reservations and subscription plans.
- Administrative Ease: Streamline administrative tasks related to managing reservations, user subscriptions, and overall parking facility operations.

1.2.2 Importance

The importance of the private parking management system lies in its ability to address the challenges associated with urban parking. Key aspects include:

- Time Efficiency: Users save time by efficiently finding and reserving parking spaces in advance.
- Revenue Generation: Through subscription plans, the system contributes to a steady revenue stream for the parking facility.
- Optimized Operations: Administrators can make data-driven decisions for better parking space allocation, leading to optimized operations.

1.2.3 Analysis of Requirements

A detailed examination of the requirements for the database application, including user expectations, system functionalities, and any constraints involved. This section explores how these requirements were identified and documented.

1.2.4 Conception Phase Using Entity-Relationship Diagram (ERD)

Presenting the Entity-Relationship Diagram (ERD) that serves as the blueprint for the database application. The ERD defines entities, attributes, relationships, and cardinalities, providing a visual representation of the database structure.

1.2.5 Data Dictionary

Inclusion of a data dictionary that offers concise descriptions of tables, attributes, and data types. This section ensures clarity in understanding each database element.

1.2.6 Integrity Constraints

A discussion on the integrity constraints applied in the database schema. Exploring how these constraints ensure data accuracy, consistency, and reliability.

1.2.7 SQL Queries and Examples

Providing SQL queries that illustrate interactions with the database. Examples cover data retrieval, insertion, and modification, explaining the purpose of each query and its relevance to the application.

1.2.8 Tools Used

Mentioning the tools and technologies employed in designing, implementing, and managing the database. A discussion on the rationale for selecting these tools and their contributions to the project.

Chapter 2

Requirement Analysis

2.1 Requirement Analysis

2.1.1 Functional Requirements

The private parking management system has been designed with a meticulous consideration of various functional requirements to provide users with a seamless and comprehensive experience.

1. **User Registration and Authentication:** The system should facilitate user account creation, ensuring a secure and straightforward registration process. User authentication mechanisms must be robust, protecting user accounts from unauthorized access.

2. **Parking Space Reservation:** Users should have the ability to search for available parking spaces based on criteria such as location, date, and duration. The reservation process should be intuitive, allowing users to select, reserve, and receive timely confirmation for their chosen parking spot.

3. **Subscription Management:** The system must support various subscription plans, including weekly, monthly, and yearly options. Users should be able to select a plan, choose a subscription start date, and receive automated notifications about upcoming subscription expirations.

4. **Reservation Editing and Deletion:** Flexibility is essential; users should be able to modify reservation details, such as date or duration, after confirmation. Additionally, a straightforward process for canceling reservations should be provided, ensuring user control and convenience.

5. **Dashboard for Users:** A personalized user dashboard is essential, displaying relevant information such as reservation history, active subscriptions, payment details, and account settings. This centralized hub enhances the overall user experience and facilitates efficient navigation.

2.1.2 Non-functional Requirements

In addition to functional aspects, non-functional requirements play a pivotal role in shaping the system's overall performance, security, and usability.

1. **Performance:** The system must exhibit exceptional performance, with rapid response times for user interactions, search queries, and reservation processing. This ensures a seamless experience, even during peak usage periods.

2. **Security:** Robust security measures are paramount. User data, including personal information and payment details, must be encrypted and stored securely. The authentication process should employ industry-standard protocols to protect user accounts from potential threats.

3. **Scalability:** To accommodate growth, the system should be designed with scalability in mind. As the number of users and parking spaces increases, the system should seamlessly adapt without compromising performance.

4. **Usability:** User interfaces should be highly intuitive, providing clear navigation and responsiveness across various devices. An emphasis on usability ensures that users, regardless of technical expertise, can interact with the system effortlessly.

5. **Reliability:** The system must be reliable, minimizing downtime and ensuring data integrity. Regular backups and failover mechanisms should be in place to mitigate potential disruptions.

2.1.3 Requirements Gathering

The requirements for this system were meticulously gathered through a multifaceted approach. The aim was to ensure a comprehensive understanding of user needs, industry practices, and specific challenges related to parking space management.

- **User Surveys:** Conducted detailed user surveys to gather insights into user expectations, pain points, and preferences regarding parking space management. Questions were tailored to elicit feedback on the current challenges users face when searching for, reserving, and managing parking spaces.

- **Interviews:** Engaged in one-on-one interviews with potential users and stakeholders, including individuals who frequently use parking facilities and those responsible for managing parking spaces. These interviews provided deeper insights into specific needs, expectations, and challenges from both user and administrative perspectives.
- **On-site Observation:** Visited a local parking facility near the university to observe the current processes and challenges faced by both users and parking space owners. Direct interaction with the owner of the parking facility provided valuable details about day-to-day operations, user interactions, and common issues faced by parking space providers.
- **Owner Feedback:** Engaged in direct communication with the owner of the parking facility to gather firsthand information about the intricacies of managing a parking space business. The owner shared insights into user behaviors, payment preferences, and the administrative aspects of handling reservations and subscriptions.

The combination of these approaches ensured a holistic and user-centric understanding of the requirements, laying the foundation for a system that addresses the unique challenges of parking space management near the university.

Chapter 3

Conception of the project

3.1 Conception Phase Using Entity-Relationship Diagram (ERD)

3.1.1 Entity-Relationship Diagram (ERD)

The Entity-Relationship Diagram (ERD) provides a visual representation of the data model for the private parking management system. This diagram illustrates the relationships between various entities and their attributes, forming the foundation for database design and implementation.

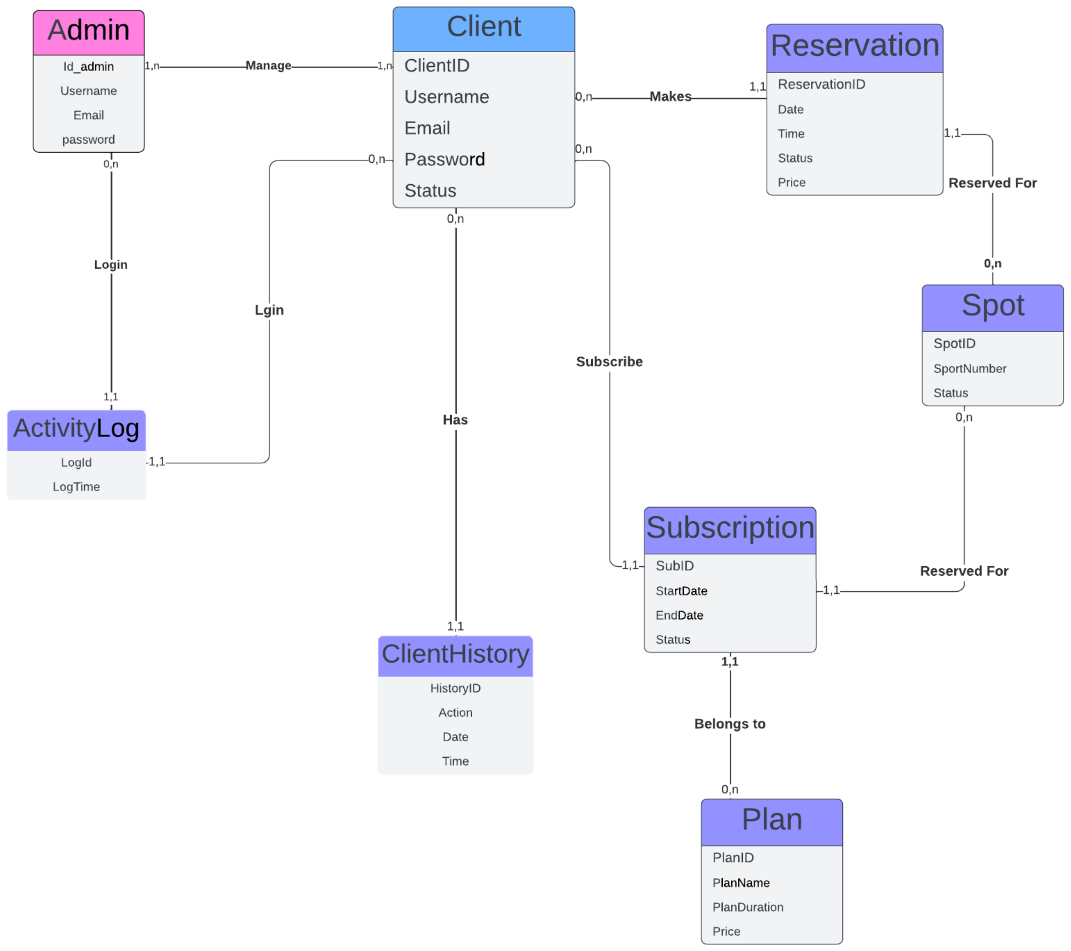


Figure 3.1: Entity-Relationship Diagram for the Private Parking Management System

3.1.2 Entities

- **Admin:** Represents an administrative user responsible for managing clients and overseeing system activities.
- **Client:** Represents an end-user who interacts with the system by making reservations and subscriptions.
- **Activity Log:** Serves as a log for recording various system activities, including login events and critical actions performed by both admins and clients.
- **Client History:** Captures historical data related to client activities, providing insights into past interactions and preferences.
- **Reservation:** Represents a specific reservation made by a client, storing details such as date, time, and the associated parking spot.
- **Subscription:** Represents a client's subscription to a particular pricing plan, including information about the plan's duration and associated parking spot.
- **Plan:** Defines different subscription plans available to clients, specifying attributes like duration and price.
- **Spot:** Represents individual parking spots within the facility available for reservation or subscription.

3.1.3 Relationships and Cardinalities

- **Admin-Manage-Client:** An admin can manage multiple clients, establishing a one-to-many (1:N) relationship.
- **Admin-Login-ActivityLog:** The login activities of an admin are recorded in the activity log, creating a one-to-many (1:N) relationship.
- **Client-Log-ActivityLog:** The login activities and interactions of a client are recorded in the activity log, forming a one-to-many (1:N) relationship.
- **Client-Makes-Reservation:** A client can make multiple reservations, resulting in a one-to-many (1:N) relationship.

- **Client-Subscribes-Subscription:** A client can have multiple subscriptions, establishing a one-to-many (1:N) relationship.
- **Subscription-Pertains-Plan:** A subscription is associated with a specific pricing plan, forming a many-to-one (M:1) relationship.
- **Subscription-Pertains-Spot:** A subscription pertains to a specific parking spot, creating a many-to-one (M:1) relationship.
- **Reservation-Pertains-Spot:** A reservation pertains to a specific parking spot, also resulting in a many-to-one (M:1) relationship.

3.1.4 Rationale

- The **Admin** entity enables centralized management of clients, ensuring effective oversight and control through the **Admin-Manage-Client** relationship.
- The **Activity Log** plays a crucial role in system security and auditing by recording login activities for both admins (**Admin-Login-ActivityLog**) and clients (**Client-Log-ActivityLog**).
- The **Client History** entity preserves historical data, facilitating personalized services and informed decision-making based on past interactions.
- The **Subscription** entity establishes relationships with both **Plan** and **Spot**, ensuring that each subscription is associated with a specific pricing plan and parking spot.
- The **Reservation** entity is linked to a **Spot**, providing details about the reserved parking location for each reservation.

3.2 Data Dictionary

Admin Table

Attribute	**Description**	**Data Type**
id-admin	Unique identifier for the admin	INT
username	Admin's username	VARCHAR
email	Admin's email address	VARCHAR
password	Admin's password (hashed)	VARCHAR

Table 3.1: Admin Table Data Dictionary

Client Table

Attribute	**Description**	**Data Type**
client-id	Unique identifier for the client	INT
username	Client's username	VARCHAR
email	Client's email address	VARCHAR
password	Client's password (hashed)	VARCHAR
status	Status of the client account	VARCHAR

Table 3.2: Client Table Data Dictionary

ActivityLog Table

Attribute	**Description**	**Data Type**
log-id	Unique identifier for the log entry	INT
login time	Time of the login event	DATETIME

Table 3.3: ActivityLog Table Data Dictionary

ClientHistory Table

Attribute	Description	Data Type
history-id	Unique identifier for the history entry	INT
action	Action performed by the client	VARCHAR
timestamp	Timestamp of the action	DATETIME

Table 3.4: ClientHistory Table Data Dictionary

Reservation Table

Attribute	Description	Data Type
reservation-id	Unique identifier for the reservation	INT
date	Date of the reservation	DATE
time	Time of the reservation	TIME
status	Status of the reservation	VARCHAR
price	Price of the reservation	DECIMAL

Table 3.5: Reservation Table Data Dictionary

Subscription Table

Attribute	Description	Data Type
subscription-id	Unique identifier for the subscription	INT
startdate	Start date of the subscription	DATE
status	Status of the subscription	VARCHAR

Table 3.6: Subscription Table Data Dictionary

Spot Table

Attribute	Description	Data Type
spot-id	Unique identifier for the parking spot	INT
spotnumber	Spot number for identification	INT
status	Status of the spot (e.g., available, reserved)	VARCHAR

Table 3.7: Spot Table Data Dictionary

Plan Table

Attribute	Description	Data Type
plan-id	Unique identifier for the plan	INT
planname	Name of the subscription plan	VARCHAR
planduration	Duration of the plan in days	INT
price	Price of the plan	DECIMAL

Table 3.8: Plan Table Data Dictionary

Manage Relationship Table

Attribute	Description	Data Type
admin-id	Foreign key referencing Admin Table	INT
client-id	Foreign key referencing Client Table	INT

Table 3.9: Manage Relationship Table Data Dictionary

Chapter 4

Development of the project

4.1 Integrity Constraints

4.1.1 Users Table

- **Primary Key Constraint:** `id` is the primary key, ensuring each user has a unique identifier.
- **Check Constraint:** `status` is checked to be either ’subscribed’ or ’regular’, ensuring valid user statuses.

4.1.2 Admin Table

- **Primary Key Constraint:** `admin_id` is the primary key, ensuring each admin has a unique identifier.

4.1.3 ClientHistory Table

- **Primary Key Constraint:** `history_id` is the primary key, ensuring each client history entry has a unique identifier.
- **Foreign Key Constraint:** `id` references the `users` table, ensuring a valid link to a user.
- **Check Constraint:** `action` is checked to be either ’subscribed’ or ’regular’.

4.1.4 Spots Table

- **Primary Key Constraint:** `spotID` is the primary key, ensuring each spot has a unique identifier.
- **Check Constraint:** `status` is checked to be either ’reserved’ or ’liberated’.

4.1.5 Plans Table

- **Primary Key Constraint:** `planID` is the primary key, ensuring each plan has a unique identifier.

4.1.6 ActivityLog Table

- **Primary Key Constraint:** `aLogID` is the primary key, ensuring each activity log entry has a unique identifier.
- **Foreign Key Constraints:** `users` and `admin` reference the `users` and `admin` tables, respectively.

4.1.7 Subscription Table

- **Primary Key Constraint:** `subscription_id` is the primary key, ensuring each subscription has a unique identifier.
- **Foreign Key Constraints:** `users`, `plans`, and `spots` reference the `users`, `plans`, and `spots` tables, respectively.
- **Check Constraint:** `status` is checked to be either 'active' or 'inactive'.

4.1.8 Notification Table

- **Primary Key Constraint:** `notification_id` is the primary key, ensuring each notification has a unique identifier.

4.1.9 Reservation Table

- **Primary Key Constraint:** `reservation_id` is the primary key, ensuring each reservation has a unique identifier.
- **Foreign Key Constraints:** `users` and `spots` reference the `users` and `spots` tables, respectively.
- **Check Constraint:** `status` is checked to be either 'active' or 'inactive'.

4.2 SQL Queries and Examples

4.2.1 Inserting a User

```
INSERT INTO users (username, passw, email, stat)
```

```
VALUES ('JohnDoe', 'password123', 'john@example.com', 'subscribed') RETURNING *;
```

4.2.2 Search Users by Name

```
SELECT * FROM users WHERE username ILIKE '%John%';
```

4.2.3 Update User Profile

```
UPDATE users SET passw = 'newpassword' WHERE id = 1 RETURNING *;
```

4.2.4 Get Admin Profile

```
SELECT * FROM admin WHERE admin_id = 1;
```

4.2.5 Create Reservation

```
INSERT INTO reservation (users, spots, resdate, restime, price, status)
VALUES (1, 1, '2023-01-06', '04:00:00', 6.00, 'active') RETURNING *;
```

4.2.6 Search Reservations by Date

```
SELECT * FROM reservation WHERE resdate::TEXT ILIKE '%2023-01-06%';
```

4.2.7 Update Reservation Status

```
UPDATE reservation SET status = 'inactive' WHERE reservation_id = 1 RETURNING *;
```

4.2.8 Update Plan

```
UPDATE plans SET planname = 'Premium Plan', planduration = 90, price = 29.99
WHERE planid = 3 RETURNING *;
```

4.2.9 Get Plan by ID

SELECT * FROM plans WHERE planid = 1;

4.2.10 Delete User

DELETE FROM users WHERE id = 1 RETURNING *;

4.2.11 Create Notificationr

INSERT INTO notification (message) VALUES ('New reservation created') RETURNING*;

4.2.12 Get All Notifications

SELECT * FROM notification;

4.2.13 Retrieving a User by Email

SELECT * FROM client WHERE email = ?; SELECT * FROM client WHERE email = 'client@gmail.com';

4.2.14 Inserting a Client History Record

INSERT INTO clientHistory (client_id, action, time) VALUES (?, ?, CURRENT_TIMESTAMP); INSERT INTO clientHistory (client_id, action, time) VALUES (34, 'reservation', CURRENT_TIMESTAMP);

4.2.15 Retrieving Client History

SELECT * FROM clientHistory WHERE client_id = ? SELECT * FROM clientHistory WHERE client_id = 16;

4.3 Query Descriptions

4.3.1 Inserting a User

- **Purpose:** This query adds a new user to the 'users' table when a user registers through the application.

- **Relevance:** It ensures accurate storage of user data, providing unique identifiers and subscription statuses.

4.3.2 Search Users by Name

- **Purpose:** This query allows searching for users based on their username, enhancing user management.
- **Relevance:** It enables administrators to efficiently find specific users.

4.3.3 Update User Profile

- **Purpose:** Used to update a user's profile information, such as changing the password.
- **Relevance:** Provides a way for users to modify their details, ensuring data accuracy and security.

4.3.4 Get Admin Profile

- **Purpose:** Retrieves the profile information of the admin with admin-id equal to 1.
- **Relevance:** Allows the admin to view and potentially update their own profile information.

4.3.5 Create Reservation

- **Purpose:** Used to create a new reservation when a user books a spot through the application.
- **Relevance:** Essential for tracking reservations, managing parking spots, and ensuring proper billing.

4.3.6 Search Reservations by Date

- **Purpose:** Enables searching for reservations based on a specific date.

- **Relevance:** Supports features like viewing reservations for a particular day, facilitating better management.

4.3.7 Update Reservation Status

- **Purpose:** Changes the status of a reservation, typically used when a reservation is completed or canceled.
- **Relevance:** Helps in keeping track of the status of reservations, aiding in effective reservation management.

4.3.8 Update Plan

- **Purpose:** Modifies details of a subscription plan, such as its name, duration, and price.
- **Relevance:** Allows the application to adapt to changes in subscription plans or correct plan information.

4.3.9 Get Plan by ID

- **Purpose:** Retrieves details of a specific subscription plan based on its planID.
- **Relevance:** Essential for displaying plan information, especially when users want to know details about their subscription.

4.3.10 Delete User

- **Purpose:** Removes a user from the system, typically used when an account is deactivated or deleted.
- **Relevance:** Ensures accurate and up-to-date user management.

4.3.11 Create Notification

- **Purpose:** Adds a new notification to the notification table, often triggered when certain events occur.

- **Relevance:** Provides a means to notify users or administrators about important events or updates within the application.

4.3.12 Get All Notifications

- **Purpose:** Retrieves all notifications from the notification table.
- **Relevance:** Supports features like displaying a notification feed, ensuring users are informed about recent activities.

4.3.13 Retrieving a User by Email

- **Purpose:** This query is used to fetch user information based on their email when a user attempts to log in through the application.
- **Relevance:** It ensures that the login process is secure and accurate by verifying the user's credentials against stored information in the database.

4.3.14 Inserting a Client History Record

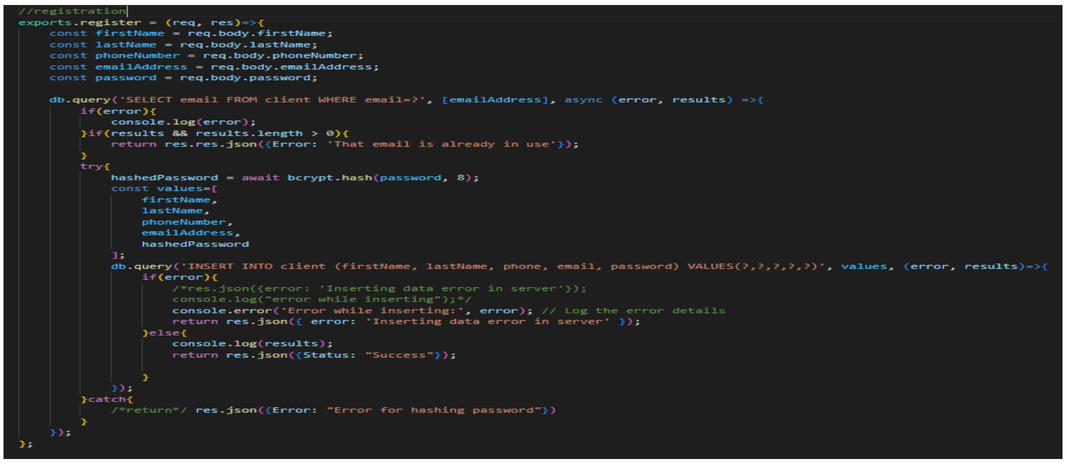
- **Purpose:** Records user actions (e.g., reservation or subscription) in the clientHistory table for historical tracking.
- **Relevance:** Provides transparency and accountability by logging user activities over time.

4.3.15 Retrieving Client History

- **Purpose:** Fetches the historical records of user actions from the clientHistory table.
- **Relevance:** Enables users to view their past activities and interactions within the application.

4.3.16 User Registration fonction

Handles the process of registering a new user by validating unique email, hashing the password, and storing user details in the client table. If the email is already in use, an error is returned; otherwise, the user is successfully registered.

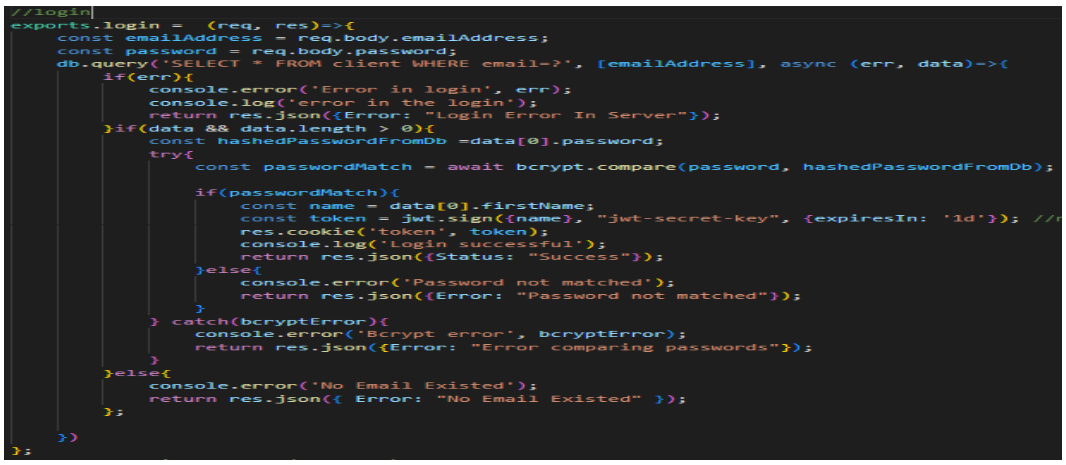


```
//registration
exports.register = (req, res)=>{
    const firstName = req.body.firstName;
    const lastName = req.body.lastName;
    const phoneNumber = req.body.phoneNumber;
    const emailAddress = req.body.emailAddress;
    const password = req.body.password;

    db.query('SELECT email FROM client WHERE email=?', [emailAddress], async (error, results) =>{
        if(error){
            console.log(error);
        }if(results && results.length > 0){
            return res.res.json({Error: 'That email is already in use'});
        }
        try{
            hashedPassword = await bcrypt.hash(password, 8);
            const values=[
                firstName,
                lastName,
                phoneNumber,
                emailAddress,
                hashedPassword
            ];
            db.query('INSERT INTO client (firstName, lastName, phone, email, password) VALUES(?,?,?,?,?)', values, (error, results)=>{
                if(error){
                    /*res.json({error: 'Inserting data error in server'});
                    console.log("error while inserting");*/
                    console.error('Error while inserting:', error); // Log the error details
                    return res.json({ error: 'Inserting data error in server' });
                }else{
                    console.log(results);
                    return res.json({Status: "Success"});

                }
            });
        }catch{
            /*return*/ res.json({Error: "Error for hashing password"})
        }
    });
};
```

Figure 4.1: User Registration fonction

4.3.17 User Login fonction

Manages the user login process by verifying the provided email and password. If the email exists and the password matches, a JWT (JSON Web Token) is generated, stored in a cookie, and sent to the client for future authentication. If the email or password is incorrect, appropriate error messages are returned.

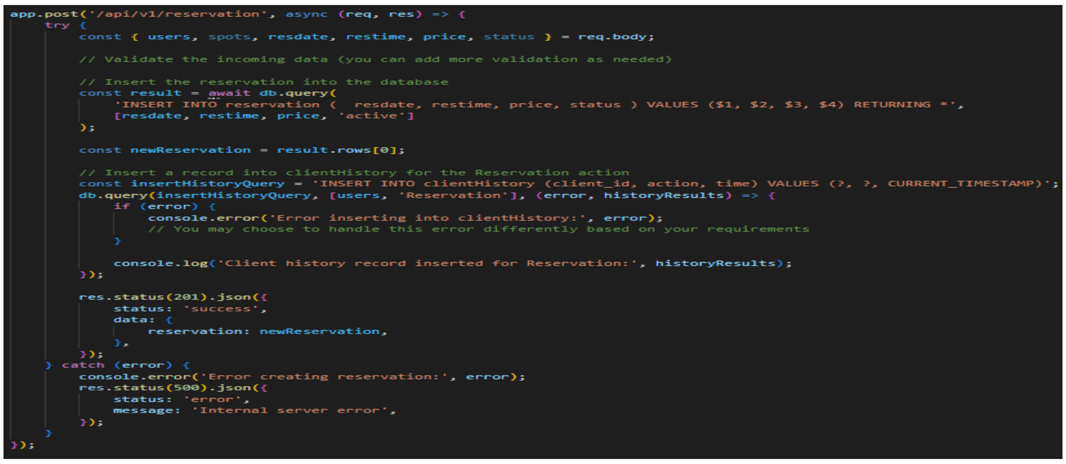


```
//login
exports.login =  (req, res)=>{
    const emailAddress = req.body.emailAddress;
    const password = req.body.password;
    db.query('SELECT * FROM client WHERE email=?', [emailAddress], async (err, data)=>{
        if(err){
            console.error('Error in login', err);
            console.log('error in the login');
            return res.json({Error: "Login Error In Server"});
        }if(data && data.length > 0){
            const hashedPasswordFromDb =data[0].password;
            try{
                const passwordMatch = await bcrypt.compare(password, hashedPasswordFromDb);

                if(passwordMatch){
                    const name = data[0].firstName;
                    const token = jwt.sign({name}, "jwt-secret-key", {expiresIn: '1d'}); //
                    res.cookie('token', token);
                    console.log('Login successful');
                    return res.json({Status: "Success"});
                }else{
                    console.error('Password not matched');
                    return res.json({Error: "Password not matched"});
                }
            } catch(bcryptError){
                console.error('Bcrypt error', bcryptError);
                return res.json({Error: "Error comparing passwords"});
            }
        }else{
            console.error('No Email Existed');
            return res.json({ Error: "No Email Existed" });
        };

    })
};
```

Figure 4.2: User Login fonction

4.3.18 Inserting a Client History Record

A record is inserted into the ’clientHistory’ table to log the reservation action, including the associated client ID, action type (’Reservation’), and timestamp. If any errors occur during the process, appropriate error messages are returned.

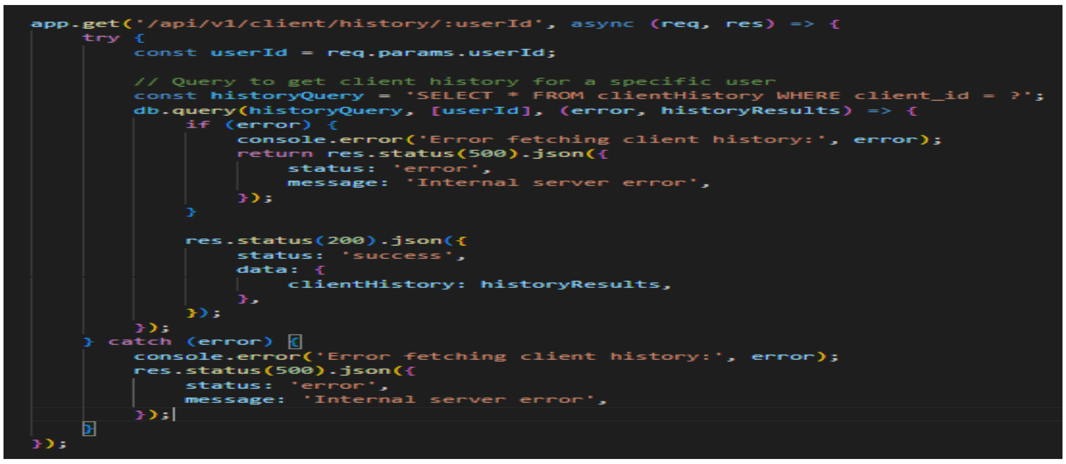


```
app.post('/api/v1/reservation', async (req, res) => {
    try {
        const { users, spots, resdate, restime, price, status } = req.body;

        // Validate the incoming data (you can add more validation as needed)

        // Insert the reservation into the database
        const result = await db.query(
            'INSERT INTO reservation (  resdate, restime, price, status ) VALUES ($1, $2, $3, $4) RETURNING *',
            [resdate, restime, price, 'active']
        );

        const newReservation = result.rows[0];

        // Insert a record into clientHistory for the Reservation action
        const insertHistoryQuery = 'INSERT INTO clientHistory (client_id, action, time) VALUES (?, ?, CURRENT_TIMESTAMP)';
        db.query(insertHistoryQuery, [users, 'Reservation'], (error, historyResults) => {
            if (error) {
                console.error('Error inserting into clientHistory:', error);
                // You may choose to handle this error differently based on your requirements
            }

            console.log('Client history record inserted for Reservation:', historyResults);
        });

        res.status(201).json({
            status: 'success',
            data: {
                reservation: newReservation,
            },
        });
    } catch (error) {
        console.error('Error creating reservation:', error);
        res.status(500).json({
            status: 'error',
            message: 'Internal server error',
        });
    }
});
```

Figure 4.3: Inserting a Client History Record

4.3.19 Client History Retrieval

Retrieves the client history for a specific user by querying the ’clientHistory’ table based on the provided user ID. If successful, the client history data is returned in the response as part of a ’success’ status. In case of any errors during the process, appropriate error messages are sent in the response with a ’500 Internal Server Error’ status.



```
app.get('/api/v1/client/history/:userId', async (req, res) => {
    try {
        const userId = req.params.userId;

        // Query to get client history for a specific user
        const historyQuery = 'SELECT * FROM clientHistory WHERE client_id = ?';
        db.query(historyQuery, [userId], (error, historyResults) => {
            if (error) {
                console.error('Error fetching client history:', error);
                return res.status(500).json({
                    status: 'error',
                    message: 'Internal server error',
                });
            }

            res.status(200).json({
                status: 'success',
                data: {
                    clientHistory: historyResults,
                },
            });
        });
    } catch (error) {
        console.error('Error fetching client history:', error);
        res.status(500).json({
            status: 'error',
            message: 'Internal server error',
        });
    }
});
```

Figure 4.4: Client History Retrieval

4.4 Tools Used

4.4.1 Communication Tools

To ensure smooth communication among the development team members, we used various tools throughout the realization of our project. The selection of these tools was guided by their ability to facilitate remote collaboration and enhance team productivity.

Among these tools, we primarily used Gmail, Messenger, and Github. **Gmail** allowed us to exchange messages and share important files, while **Messenger** was used for instant discussions and quick decision-making. Finally, **Github** was employed to manage the source code and pull requests. It served as a platform for code commenting, tracking changes, sharing code, and handling conflicts.

In addition to Gmail, Messenger, and Github, we also utilized **Google Meet** for virtual team meetings. Meet enabled us to conduct remote meetings to discuss progress, address issues, and plan tasks. The use of this tool facilitated regular communication among team members, even when geographically distant.

Meet presented numerous advantages for our project. It facilitated exchanges and decision-making by enabling real-time discussions with screen and file sharing capabilities. Additionally, it maintained a comprehensive history of meetings, proving useful for issue resolution and progress tracking.

The use of these tools offered numerous benefits for our project. They facilitated coordination among team members, providing real-time tracking of project progress and tasks. Furthermore, they allowed for storing a complete history of conversations, proving valuable for tracking changes and resolving conflicts.

In summary, the use of these tools significantly contributed to the success of our project by improving team collaboration and productivity.

Figure 4.5: Logo de Google meet

Figure 4.6: Logo de Messenger

Figure 4.7: Logo de Github

Figure 4.8: Logo de Gmail

4.4.2 Usage of the VS IDE (Visual Studio)

For the development of the project, we used Microsoft's IDE VS (Visual Studio), which is a highly popular Integrated Development Environment (IDE) for creating web applications. VS provides a wide range of features and components to facilitate development, debugging, project management, and collaboration among team members. We particularly appreciated the integration of VS with the Git version control system, allowing us to track code changes, work simultaneously on the same files, and manage conflicts easily. The VS IDE (Visual Studio) was very helpful for several reasons during the development of our project. Here are some ways in which this tool assisted us:

- Faster Development: VS offers numerous features to accelerate web application development, such as project templates, refactoring tools, keyboard shortcuts, etc. This allowed us to code more quickly and efficiently.

- Easy Debugging: VS has a built-in debugger that helps quickly detect and correct code errors. We also used breakpoints to suspend program execution at specific points to better understand errors.

- Streamlined Collaboration: VS provides collaboration tools that allowed our team to work more efficiently together, including code comments, change notifications, and issue tracking functions.

4.4.3 Design Tool

We used Figma, an online graphic design tool, to create wireframes and prototypes for our project. Figma allowed us to collaborate in real-time and work efficiently on designs with team members. We could quickly iterate on designs based on feedback from the team and users, taking into account user experience (UX) principles to create a user-friendly and easy-to-navigate interface.

Figure 4.9: figma.

4.4.4 Front-end Tools

We chose to use the **React.js** framework for creating the user interface of our project. React.js is a popular open-source JavaScript library used for building dynamic and reactive user interfaces. We opted for React.js due to its efficient management of component state, component reusability, and data binding, ensuring a smooth and responsive user experience.

Implementing React.js in our project involved creating various reusable components, managing component states, and data binding to ensure a seamless user experience. This component-based approach allowed us to work more modularly, dividing the user interface into autonomous and easily modifiable parts, enhancing team efficiency.

In addition to using React.js, we also employed two other tools in the development of our project: **Vite and Tailwind CSS.**

Vite is a fast and lightweight build tool for modern web applications. It is designed to accelerate the development process by providing features such as hot reloading and quick compilation. Vite was chosen for its speed and efficiency, enabling rapid development and testing of our application.

Tailwind CSS, on the other hand, is a utility-first CSS library that allowed us to quickly style our application without writing custom CSS. With Tailwind, we could easily apply predefined CSS classes to style our user interface, saving time and allowing us to focus more on the application's features.

By combining the use of React.js, Vite, and Tailwind, we were able to develop our web application quickly and efficiently, delivering a responsive and well-styled user experience.

Figure 4.10: Logo de tailwind

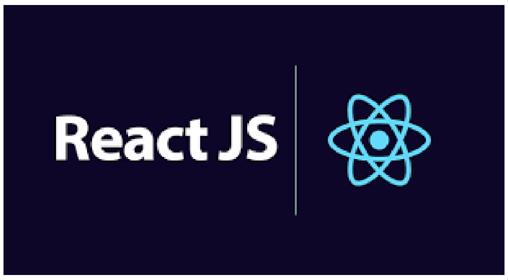


Figure 4.11: Logo de react.js

Figure 4.12: Logo de vite

4.4.5 Backend Tools

Programming Language

JavaScript is the primary language utilized for both server-side and client-side development in our project.

Server-Side Framework

For constructing the server, we employed **Node.js** in conjunction with the **Express.js** framework. Node.js serves as a JavaScript runtime for server-side development, and Express.js is a web application framework offering a robust set of features for building web and mobile applications.

Database

Our database operations were orchestrated using **PostgreSQL**. SQL queries were employed for interacting with the database, ensuring efficient and secure data storage and retrieval.

HTTP Requests

We opted for the **Axios** library to handle HTTP requests. Axios simplifies the process of sending asynchronous HTTP requests to the server, making it an essential tool for client-server communication.

API Testing Tool

POSTMAN was our chosen testing tool for API development. It simplifies tasks such as building, testing, and documenting APIs and HTTP requests. POSTMAN provides a user-friendly interface for testing different endpoints, managing environments, and validating API responses.

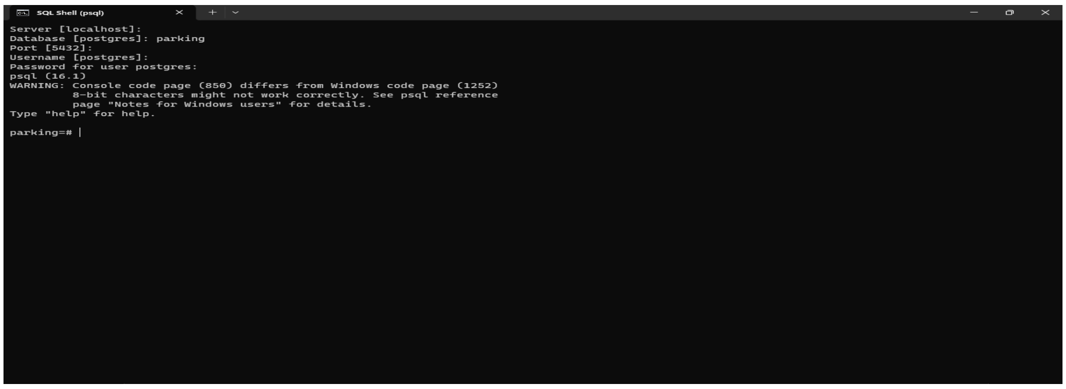
Additional Libraries and Tools

- **dotenv:** Used for environment variable management, enabling a secure and organized approach to handling sensitive information like API keys or database credentials.

- **cors:** Implemented for handling Cross-Origin Resource Sharing (CORS), ensuring secure communication between the frontend and backend, especially when hosted on different domains.

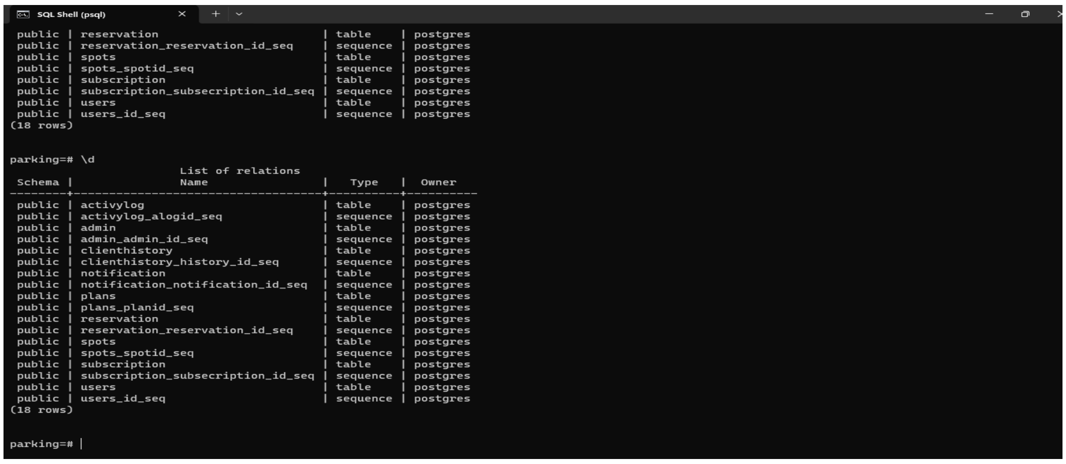
- **morgan:** Employed for HTTP request logging. Morgan logs detailed information about each incoming request, facilitating debugging and monitoring of server activities.

These backend tools collectively contribute to the efficient development, testing, and maintenance of our web application.



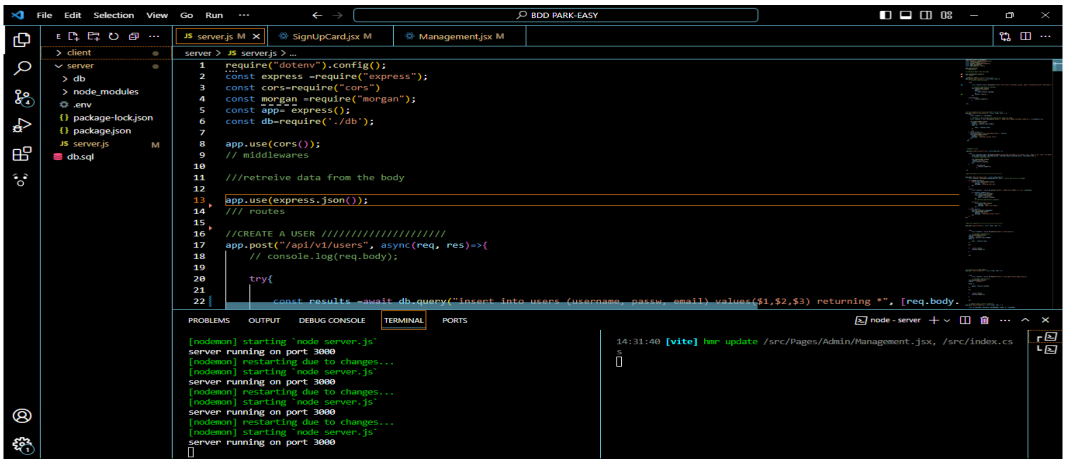
```
Server [localhost]:
Database [postgres]: parking
Port [5432]:
Username [postgres]:
Password for user postgres:
psql (16.1)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

parking=# |
```

Figure 4.13: Access the database using SQL shell.



```
 public | reservation                          | table    | postgres
 public | reservation_reservation_id_seq       | sequence | postgres
 public | spots                                | table    | postgres
 public | spots_spotid_seq                     | sequence | postgres
 public | subscription                         | table    | postgres
 public | subscription_subsecription_id_seq    | sequence | postgres
 public | users                                | table    | postgres
 public | users_id_seq                         | sequence | postgres
(18 rows)


parking=# \d
                    List of relations
 Schema |                 Name                 |   Type   |  Owner
--------+--------------------------------------+----------+----------
 public | activylog                            | table    | postgres
 public | activylog_alogid_seq                 | sequence | postgres
 public | admin                                | table    | postgres
 public | admin_admin_id_seq                   | sequence | postgres
 public | clienthistory                        | table    | postgres
 public | clienthistory_history_id_seq         | sequence | postgres
 public | notification                         | table    | postgres
 public | notification_notification_id_seq     | sequence | postgres
 public | plans                                | table    | postgres
 public | plans_planid_seq                     | sequence | postgres
 public | reservation                          | table    | postgres
 public | reservation_reservation_id_seq       | sequence | postgres
 public | spots                                | table    | postgres
 public | spots_spotid_seq                     | sequence | postgres
 public | subscription                         | table    | postgres
 public | subscription_subsecription_id_seq    | sequence | postgres
 public | users                                | table    | postgres
 public | users_id_seq                         | sequence | postgres
(18 rows)


parking=# |
```

Figure 4.14: Part of the database schema

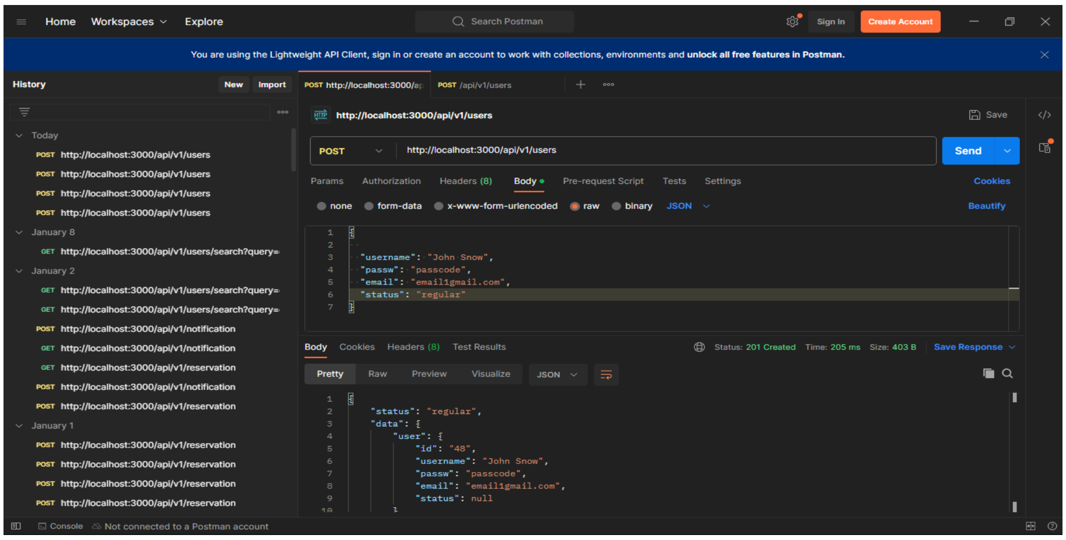


```
require("dotenv").config();
const express =require("express");
const cors=require("cors")
const morgan =require("morgan");
const app= express();
const db=require('./db');

app.use(cors());
// middlewares

///retreive data from the body

app.use(express.json());
/// routes

//CREATE A USER //////////////////////
app.post("/api/v1/users", async(req, res)=>{
    // console.log(req.body);

    try{

        const results =await db.query("insert into users (username, passw, email) values($1,$2,$3) returning *", [req.body.
```

```
[nodemon] starting `node server.js`
server running on port 3000
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
server running on port 3000
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
server running on port 3000
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
server running on port 3000
```

```
14:31:40 [vite] hmr update /src/Pages/Admin/Management.jsx, /src/index.cs
s
```

Figure 4.15: Interface to create a server



POST http://localhost:3000/api/v1/users

```
{

  "username": "John Snow",
  "passw": "passcode",
  "email": "email1gmail.com",
  "status": "regular"
}
```

Status: 201 Created Time: 205 ms Size: 403 B

```
{
    "status": "regular",
    "data": {
        "user": {
            "id": "48",
            "username": "John Snow",
            "passw": "passcode",
            "email": "email1gmail.com",
            "status": null
        }
```

History

```
Today
  POST http://localhost:3000/api/v1/users
  POST http://localhost:3000/api/v1/users
  POST http://localhost:3000/api/v1/users
  POST http://localhost:3000/api/v1/users
January 8
  GET  http://localhost:3000/api/v1/users/search?query=
January 2
  GET  http://localhost:3000/api/v1/users/search?query=
  GET  http://localhost:3000/api/v1/users/search?query=
  POST http://localhost:3000/api/v1/notification
  GET  http://localhost:3000/api/v1/notification
  GET  http://localhost:3000/api/v1/reservation
  POST http://localhost:3000/api/v1/notification
  POST http://localhost:3000/api/v1/reservation
January 1
  POST http://localhost:3000/api/v1/reservation
  POST http://localhost:3000/api/v1/reservation
  POST http://localhost:3000/api/v1/reservation
  POST http://localhost:3000/api/v1/reservation
```

You are using the Lightweight API Client, sign in or create an account to work with collections, environments and unlock all free features in Postman.

Figure 4.16: Postman interface to test API for creating a user

4.5 Presentation of Graphic Interfaces

4.5.1 Presentation of Graphic Interfaces

Our project includes a crucial component: the interface, whether it is intended for end-users or developers. In this section, we will present the different interfaces that we have developed.

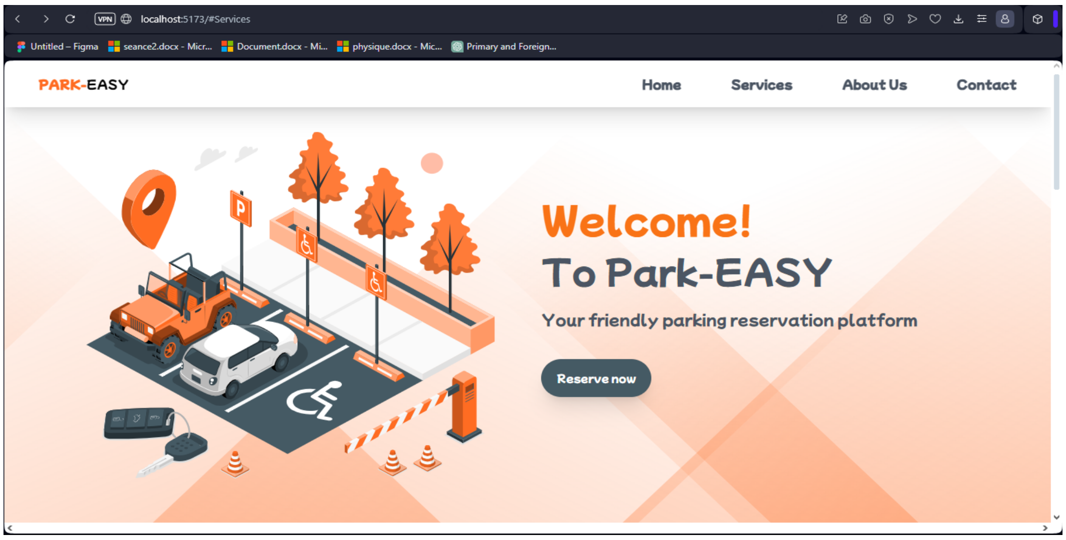
-Homepage

The homepage is often the first page that visitors see when they access your website.Therefore,it should be welcoming, attractive, and easy to navigate.

Welcome: a welcoming message to your visitors.

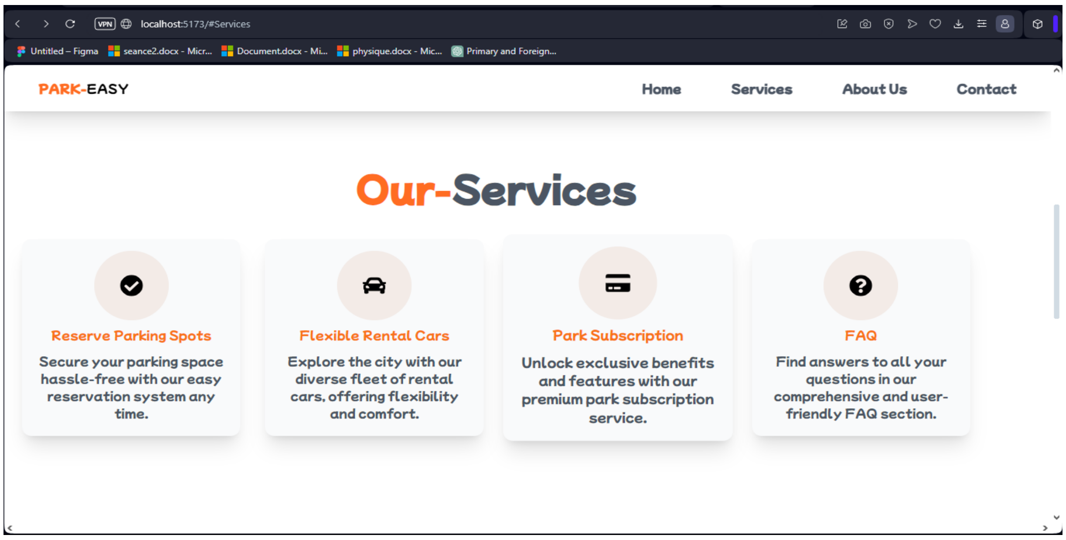
Services: features of our project.

Contact: contact information such as an email address, phone number.



Welcome!
To Park-EASY
Your friendly parking reservation platform

Figure 4.17: Homepage - Welcome



Our-Services

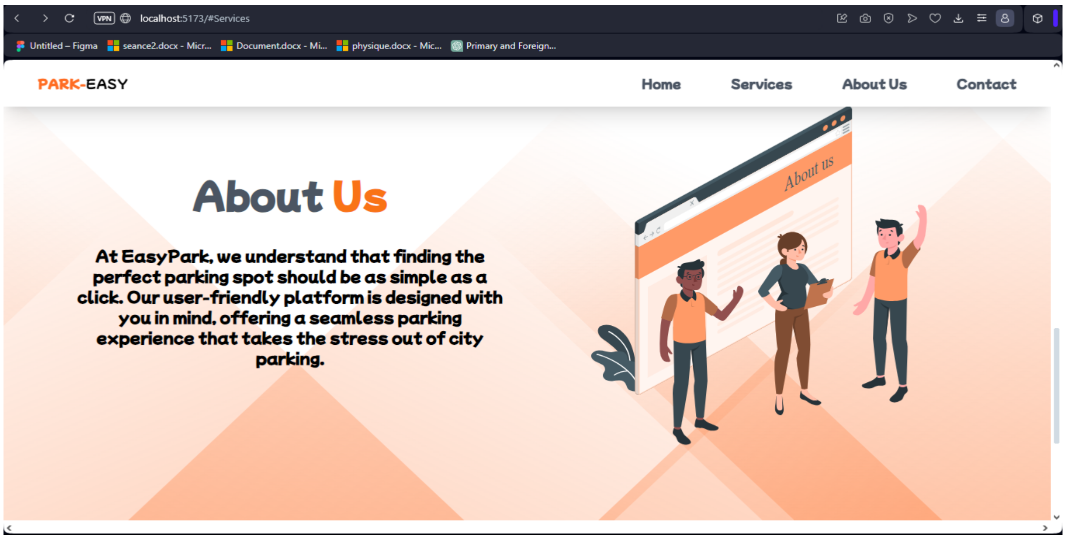
Reserve Parking Spots
Secure your parking space hassle-free with our easy reservation system any time.

Flexible Rental Cars
Explore the city with our diverse fleet of rental cars, offering flexibility and comfort.

Park Subscription
Unlock exclusive benefits and features with our premium park subscription service.

FAQ
Find answers to all your questions in our comprehensive and user-friendly FAQ section.

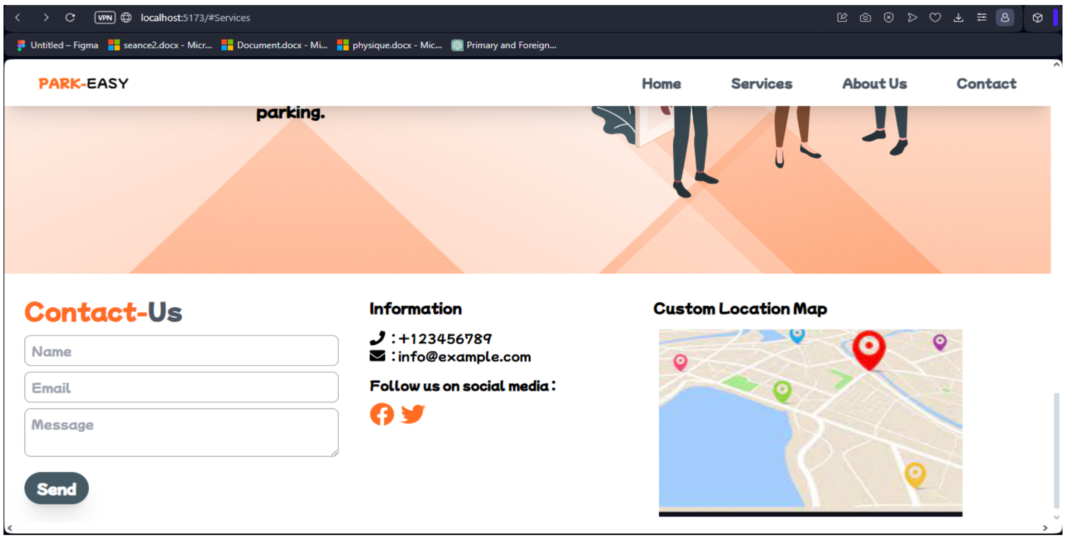
Figure 4.18: Homepage - Services



About Us

At EasyPark, we understand that finding the perfect parking spot should be as simple as a click. Our user-friendly platform is designed with you in mind, offering a seamless parking experience that takes the stress out of city parking.

Figure 4.19: Homepage - About Us



Contact-Us

Name

Email

Message

Send

Information

:+123456789

:info@example.com

Follow us on social media :

Custom Location Map

Figure 4.20: Homepage - contact

When the user clicks ’Reserve now’ they are redirected directly to the login page.

-Authentification

In our car parking website, the user needs to authenticate to access features reserved for clients or administrators. Once authenticated,they will have access to different functionalities.

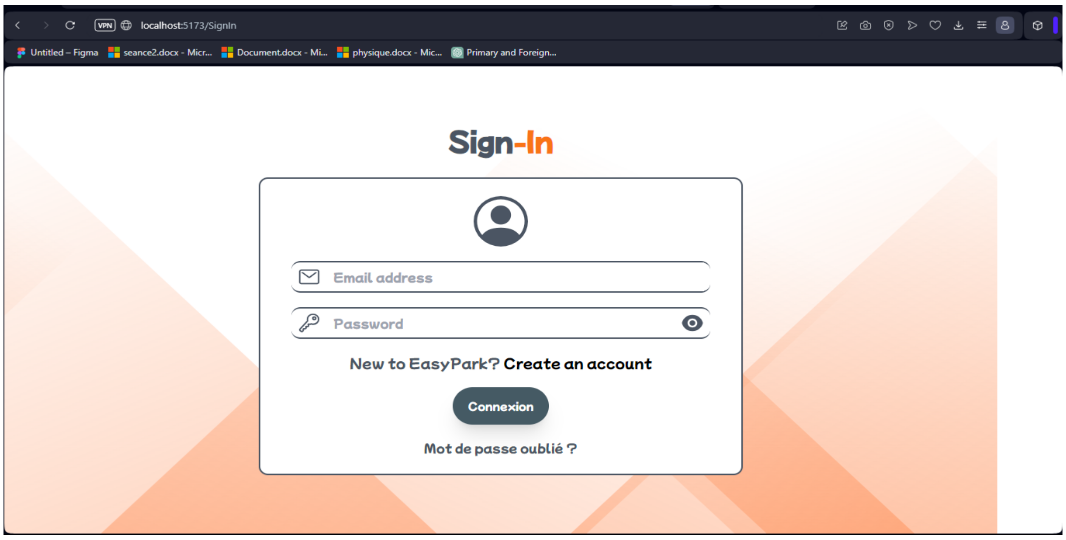


Figure 4.21: Authentification-login

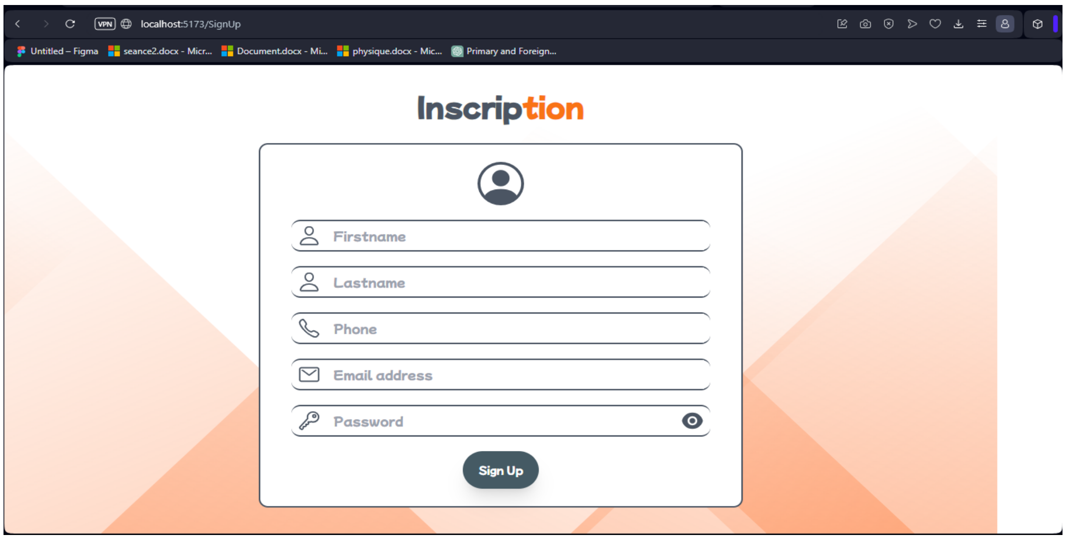
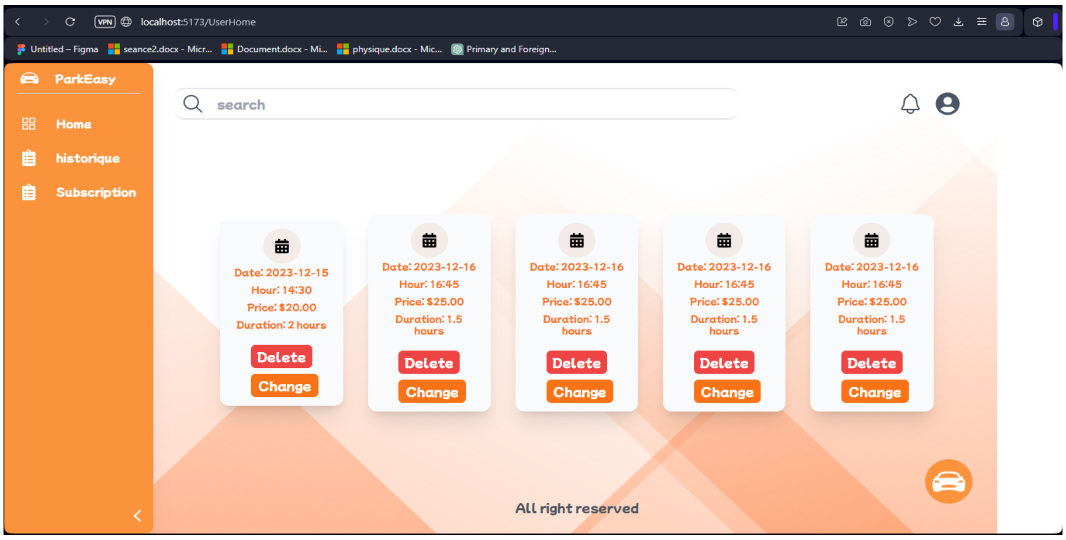


Figure 4.22: Authentification-Sign-Up

-Client Interface Overview:

The client interface is designed to provide a seamless and user-friendly experience for individuals using the car parking services. Users can easily navigate through various features, including making reservations, selecting subscription plans, editing their profiles, and reviewing the history of past reservations. The interface is intuitive, offering a visually appealing layout that ensures clients can efficiently manage their parking needs. The following screenshots showcase the key components and functionalities of the client interface.



Date: 2023-12-15
Hour: 14:30
Price: $20.00
Duration: 2 hours

Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

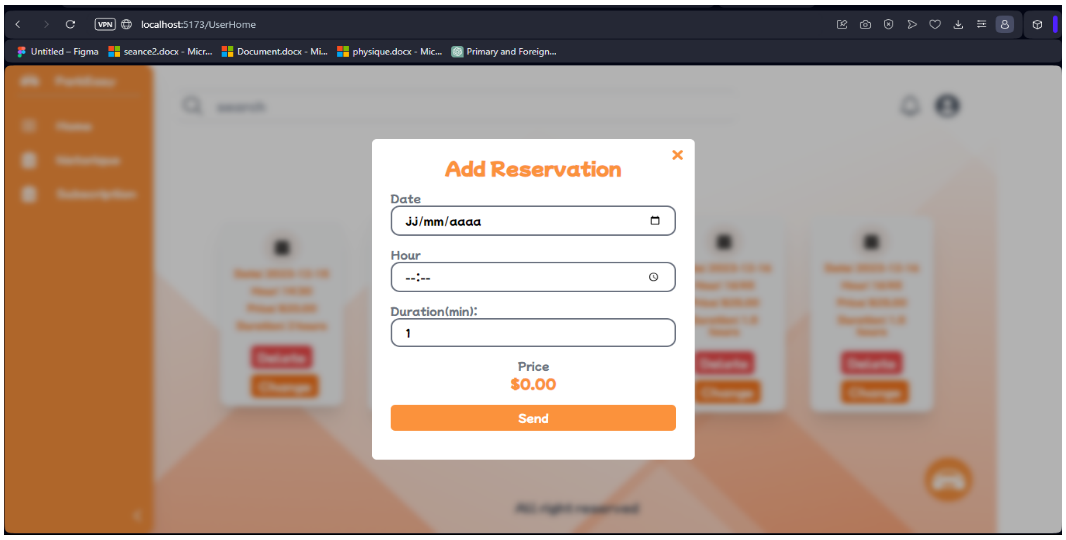
Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

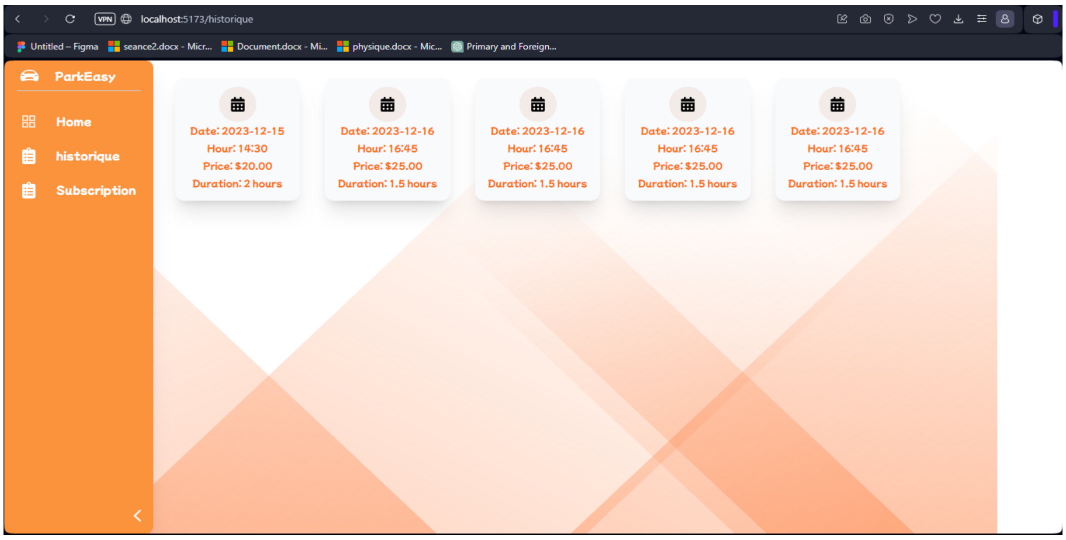
Figure 4.23: Client Interface - Home

The Reservation Service interface provides clients with a seamless experience to reserve parking spots.Users can easily select their preferred time and date, and confirm the reservation.This intuitive interface aims to simplify the process of securing a parking space,ensuring a hassle-free experience for clients.Clients can confidently make, modify, or cancel reservations, contributing to efficient parking management.



Date
jj/mm/aaaa
Hour
--:--
Duration(min):
1
Price
$0.00

Figure 4.24: Client Interface - Reservation Service



Date: 2023-12-15
Hour: 14:30
Price: $20.00
Duration: 2 hours

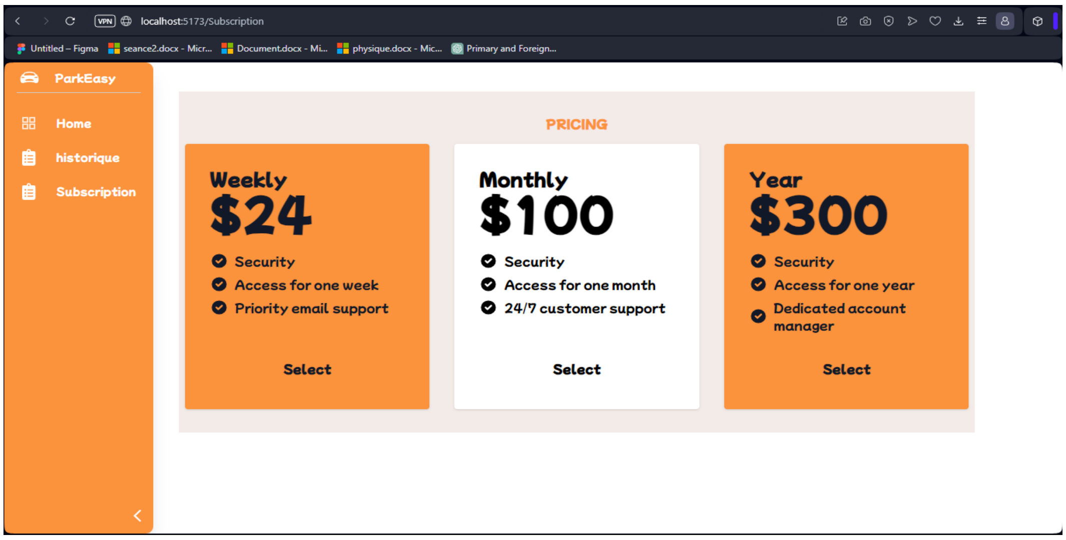
Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

Date: 2023-12-16
Hour: 16:45
Price: $25.00
Duration: 1.5 hours

Figure 4.25: Client Interface - History of Past Reservations



Weekly
$24
- Security
- Access for one week
- Priority email support

Select

Monthly
$100
- Security
- Access for one month
- 24/7 customer support

Select

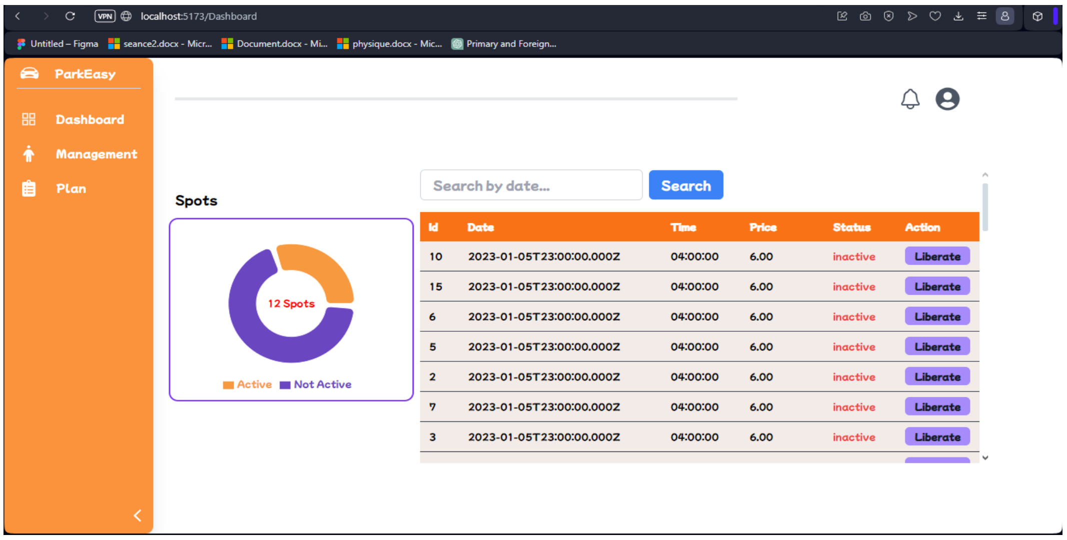
Year
$300
- Security
- Access for one year
- Dedicated account manager

Select

Figure 4.26: Client Interface - Select a Subscription

-Admin Interface Overview:

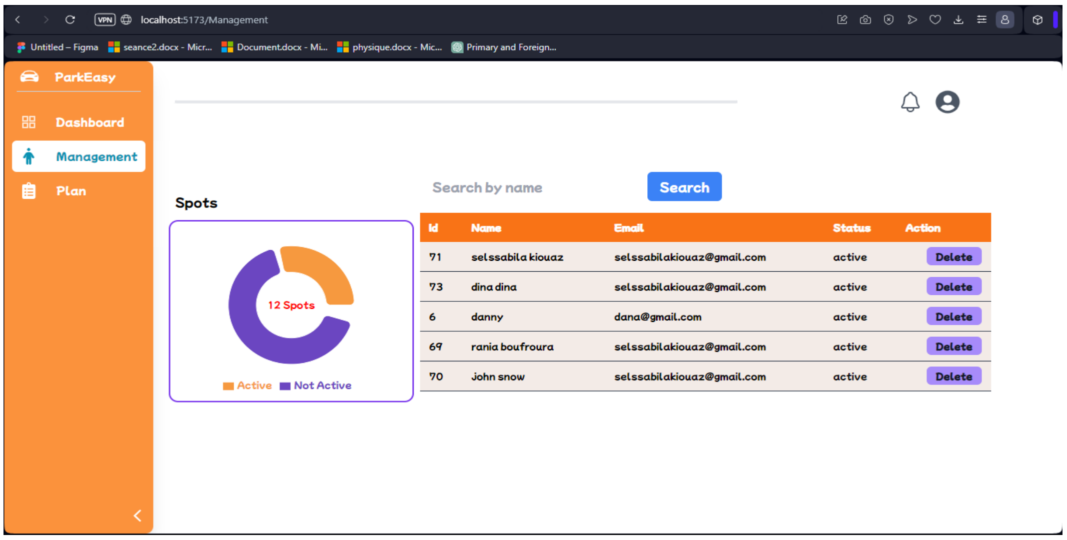
The client interface is designed to provide a seamless and user-friendly experience for individuals using the car parking services. Users can easily navigate through various features, including making reservations, selecting subscription plans, editing their profiles, and reviewing the history of past reservations. The interface is intuitive, offering a visually appealing layout that ensures clients can efficiently manage their parking needs. The following screenshots showcase the key components and functionalities of the client interface.



Id	Date	Time	Price	Status	Action
10	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
15	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
6	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
5	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
2	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
7	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
3	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate

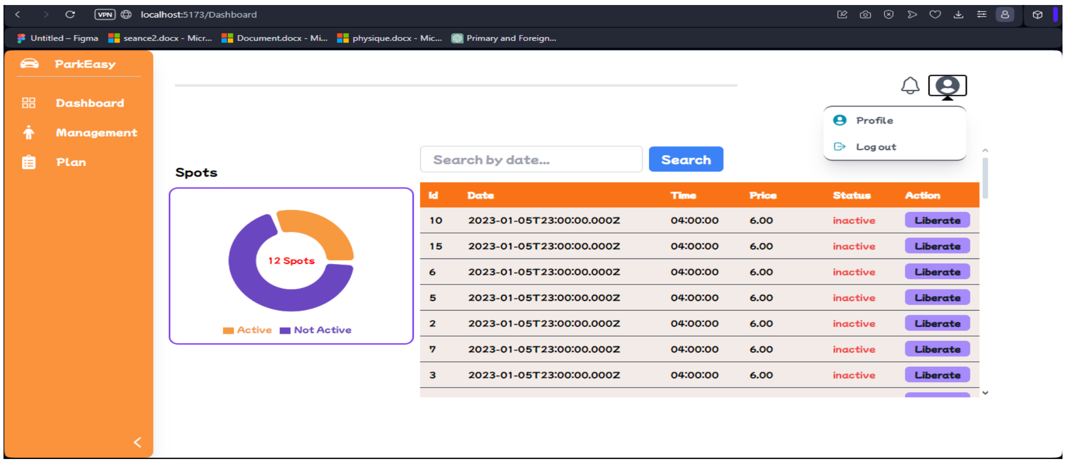
Figure 4.27: Admin-Dashboard(Spots)

Admins can efficiently view or deactivate user accounts as needed. This interface provides a comprehensive overview of user details, allowing administrators to easily search for specific users based on various parameters.This streamlined user management system contributes to effective administration and enhances overall system security.



Id	Name	Email	Status	Action
71	selssabila kiouaz	selssabilakiouaz@gmail.com	active	Delete
73	dina dina	selssabilakiouaz@gmail.com	active	Delete
6	danny	dana@gmail.com	active	Delete
69	rania boufroura	selssabilakiouaz@gmail.com	active	Delete
70	john snow	selssabilakiouaz@gmail.com	active	Delete

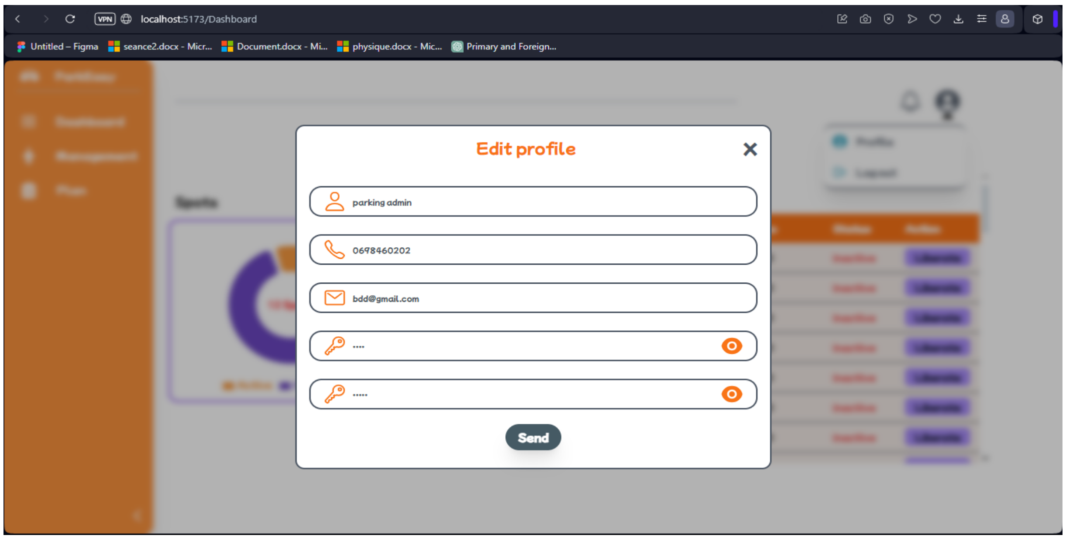
Figure 4.28: Admin-Dashboard(User management)



Id	Date	Time	Price	Status	Action
10	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
15	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
6	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
5	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
2	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
7	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
3	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate

Figure 4.29: Admin-Profile

Admin Interface - Edit Profile: The Admin Edit Profile section provides administrators with the capability to modify their user profile information. This includes essential details such as name, contact information, and other relevant credentials. Administrators can easily update their profile to ensure accurate and up-to-date identification within the system. This feature enhances the security and accountability of administrative actions by maintaining accurate user information.



parking admin

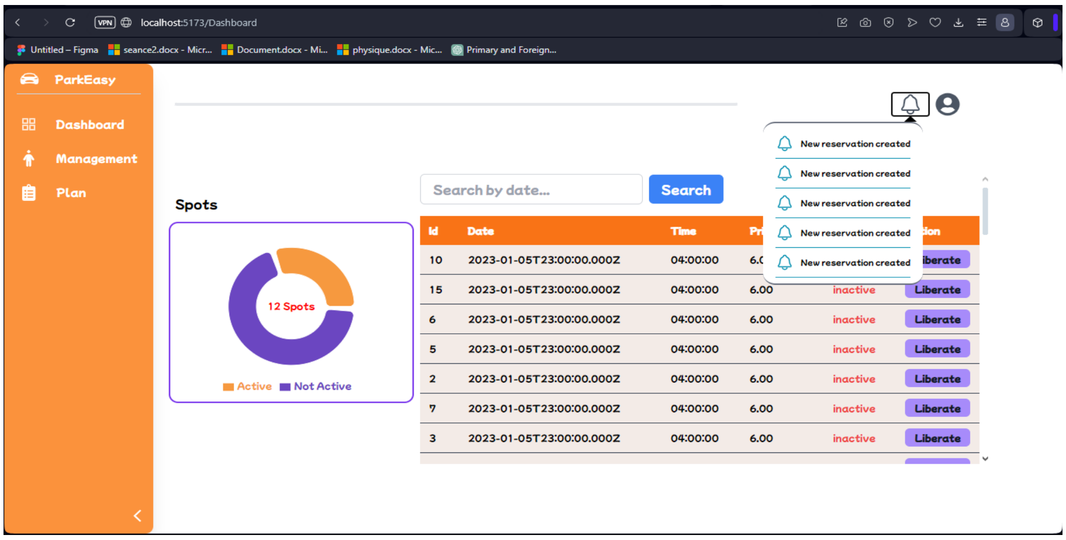
0698460202

bdd@gmail.com

....

.....

Figure 4.30: Admin-Edit profile



New reservation created

New reservation created

New reservation created

New reservation created

New reservation created

Id	Date	Time	Pr		ion
10	2023-01-05T23:00:00.000Z	04:00:00	6.0		iberate
15	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
6	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
5	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
2	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
7	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate
3	2023-01-05T23:00:00.000Z	04:00:00	6.00	inactive	Liberate

Figure 4.31: Admin-notification

Chapter 5

Conclusion

5.1 Conclusion

In the course of our Database Management System (BDD) module, the development of the parking management project has been a practical and illuminating journey. This endeavor was not merely about creating a reservation system but, fundamentally, about leveraging database principles to streamline parking operations.

The adoption of **PostgreSQL** as our database engine, complemented by the use of **Node.js** and **Express.js** for server-side logic, has provided a robust foundation for our parking system. SQL queries seamlessly integrated into our Node.js server not only ensured efficient data retrieval and manipulation but also aligned with the relational nature of our data.

Our choice of **Axios** for HTTP requests and **POSTMAN** for API testing demonstrated a commitment to robust communication channels. Furthermore, the incorporation of additional tools like **dotenv** for environment variable management, **cors** for handling Cross-Origin Resource Sharing, and **morgan** for HTTP request logging contributed to a well-rounded and secure system.

The frontend, powered by **React.js**, **Vite**, and **Tailwind CSS**, not only delivered a responsive and visually appealing user interface but also showcased the seamless integration of frontend and backend technologies.

In conclusion, this project was more than a learning experience; it was a practical application of database concepts in real-world problem-solving. It underscores the pivotal role databases play in managing, organizing, and optimizing data. As we conclude this module, our parking project stands as a testament to our ability to integrate theoretical knowledge with practical skills, offering a glimpse into the future of intelligent and data-driven parking management solutions.