CENG364

Database Applications

Assist. Prof. Dr. Sedef Demirci

Muhammet Buğra GÜLER – Emrehan ÇELİK

201180003 - 201180036

E-commerce System Based on Book Shopping

09.01.2024

E-commerce System Based on Book Shopping

Group Members (ID and Name): 201180003 - Muhammet Buğra Güler

201180036 - Emrehan Çelik

Project Title: E-commerce System Based on Book Shopping

Project Summary: This project aims to create an e-commerce system focused on book shopping. Users will be able to search, browse, and purchase books online. The project will utilize C#.NET Framework Windows Forms [1] for creating the graphical user interface, express.js [4] with VS Code [2] for the backend, and MySQL [3] for the database. We used the Kitapyurdu website to get book information. [5]

Introduction

The primary goal of our project is to create a comprehensive and user-friendly book management and e-commerce platform. Our motivation stems from the need for a more integrated and feature-rich system compared to existing book management systems. While there are several platforms available for similar purposes, our system stands out due to its advanced functionalities and user-centric design

System Requirements and Technologies

1. Database Technology: Utilizes an RDBMS(MySQL) for data storage and management.

2. Programming Languages: Employs node.js for backend development, complemented by a C# frontend.

Work distribution was conducted in the following manner:

- Emrehan was responsible for the backend coding and integration of authentication, security, and data related to books and users within the system.

- Buğra handled the coding of the user interface, as well as the implementation of payment processing, adding ratings, adding comments, and managing order operations.

System Modules and Functional Descriptions

1. Authentication Module

-Description: Ensures secure user access.

-Functionality: Manages user authentication and session control.

2. Payment Module

- Description: This module provides secure and efficient payment operations within the system.

- Functionality: It handles all aspects of payment processing, including the initiation of transactions, verification of payment details, and confirmation of successful transactions. The module is designed to support multiple payment methods, ensuring a convenient and flexible user experience.

3. Book Management Module (CRUD Operations)

-Description: Central module for book operations.

- Functionality: Facilitates creating, viewing, updating, and deleting book records.

4. User Management Module (CRUD Operations)

- Description: Manages user profiles.

- Functionality: Includes editing user profiles and managing user data.

5. Rating and Review Module

- Description: Allows user interaction.

- Functionality: Enables book rating and review posting.

6. Order, Comment, Address Modules

- Description: Manages orders, user comments, and address details.

- Functionality: Handles the processing of orders, managing comments, and storing user address information.

Each module is meticulously crafted to ensure an engaging and efficient user experience, while maintaining the robustness and reliability of the system.

SQL Design

As seen in Figure 1.1, we have the fields present in the book table. Additionally, we have foreign key relationships with the Genre, Language, Author, and Publisher tables.

Figure 1.1 (Book Table and its foreign key tables)

In Figure 1.2, we see the DDL queries for the 'Book' table and its foreign key tables. ISBN has been set as a unique key in our example because it is a distinguishing characteristic of a book. Additionally, with default configuration, when inserting values into the 'number of sales' and 'stock quantity' fields, we provide default value assignment when a null value is entered. Naturally, foreign key fields have been set as 'not null'.

```sql
CREATE TABLE `Book`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `title` VARCHAR(100) NOT NULL,
    `author` INT UNSIGNED NOT NULL,
    `ISBN` VARCHAR(13) NOT NULL,
    `price` DECIMAL(8, 2) NOT NULL,
    `stock_quantity` INT UNSIGNED NOT NULL DEFAULT '10',
    `publisher` INT UNSIGNED NULL,
    `publication_date` DATE NULL,
    `number_of_sales` INT UNSIGNED NOT NULL DEFAULT '0',
    `number_of_page` INT UNSIGNED NULL,
    `description` TEXT NULL,
    `genre` INT UNSIGNED NULL,
    `language` INT UNSIGNED NULL,
    `image` BLOB(153600) NULL
);
ALTER TABLE `Book` ADD UNIQUE (`ISBN`);
CREATE TABLE `Publisher`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `publisher_name` VARCHAR(100) NOT NULL
);
CREATE TABLE `Genre`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `genre_name` VARCHAR(50) NOT NULL
);
CREATE TABLE `Language`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `language_name` VARCHAR(50) NOT NULL
);
CREATE TABLE `Author`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `author_name` VARCHAR(50) NOT NULL
);
```

Figure 1.2 (Showing SQL DDL query to form Books and its foreign key tables)

As seen in Figure 1.3, we can observe the relationships we've designed for the 'adding comment' and 'rating' features added to the system. Additionally, the 'user' table for authentication processes can be seen in our design. We differentiate between managers and customers using the 'is_manager' field, and by moving customer IDs to a separate 'customer' table, we make the design more manageable.

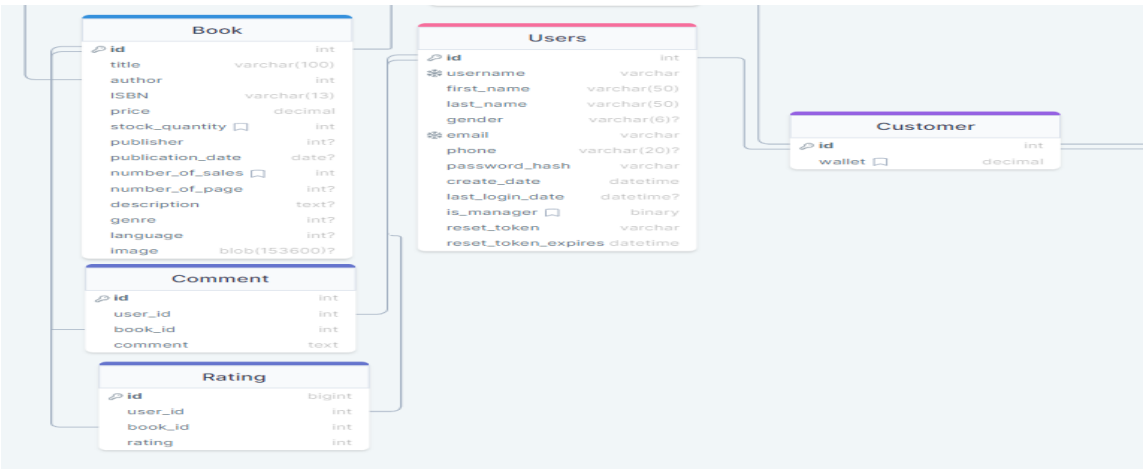


Figure 1.3 (showing relationship for comment, rating operations and user and customer tables for authentication operations)

As seen from Figure 1.4, for authentication and 'adding comment' and 'adding rating' processes, we have employed various custom configurations. Setting 'email' and 'username' as unique keys is necessary for secure authentication processes. We manage forget password operations using 'resettoken_expires' and 'reset token' fields. We execute the session management operation with the 'last_login_date' field, which is configured using JWT. After applying the hash operation using the bcrypt library to the 'password_hash' field, we store the modified password values.

```
CREATE TABLE `Users`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `username` VARCHAR(255) NOT NULL,
    `first_name` VARCHAR(50) NOT NULL,
    `last_name` VARCHAR(50) NOT NULL,
    `gender` VARCHAR(6) NULL,
    `email` VARCHAR(255) NOT NULL,
    `phone` VARCHAR(100) NULL,
    `password_hash` VARCHAR(255) NOT NULL,
    `create_date` DATETIME NOT NULL,
    `last_login_date` DATETIME NULL,
    `reset_Token` VARCHAR(255) NULL,
    `reset_Token_Expires` DATETIME NULL,
    `is_manager` BIT NOT NULL DEFAULT 0
);
ALTER TABLE
    `Users` ADD UNIQUE `users_username_unique`(`username`);
ALTER TABLE
    `Users` ADD UNIQUE `users_email_unique`(`email`);
CREATE TABLE `Rating`(
    `id` BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `user_id` INT UNSIGNED NOT NULL,
    `book_id` INT UNSIGNED NOT NULL,
    `rating` INT UNSIGNED NOT NULL
);
CREATE TABLE `Comment`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `user_id` INT UNSIGNED NOT NULL,
    `book_id` INT UNSIGNED NOT NULL,
    `comment` TEXT NULL
```

Figure 1.4 (Showing SQL DDL queries to form rating, comment, and user tables)

The 'order details' table, as seen in Figure 1.5, is used for adding items to the cart. The 'order' table stores data after the order placement process. We associate the 'shopping address' in the order table with the address table held by each user in the system. Similarly, we maintain the necessary fields for the 'order' table using foreign key relationships with the 'payment method' and 'order status' tables.

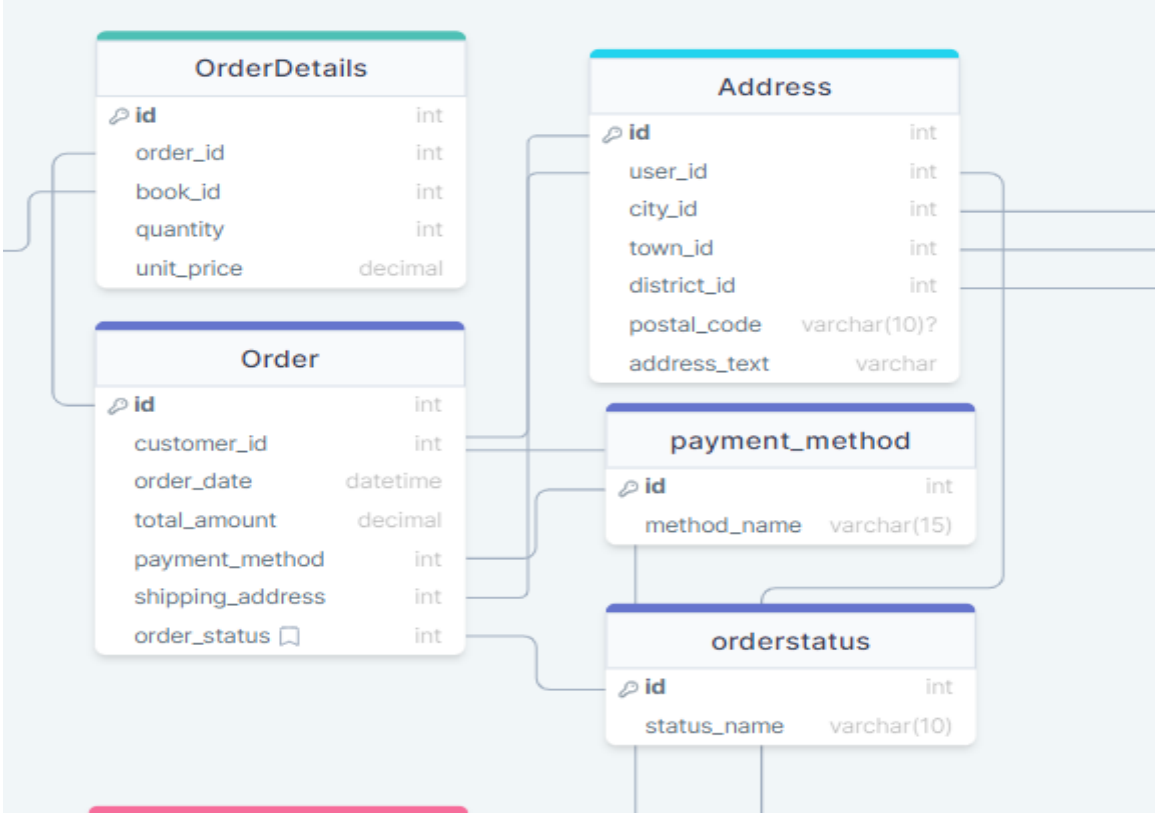


Figure 1.5 (showing the tables and relationship with each other to manage order processing)

As shown in Figure 1.6, we can see the DDL codes for the tables used to manage order processes. What I would like to mention is that we did not use any foreign key constraints in our DDL codes. We managed these operations in the application layer. Like the examples above, we have used a default order status structure, and this data corresponds to the 'preparing' status defined in our 'order status' table.

```sql
CREATE TABLE `Order`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `customer_id` INT UNSIGNED NOT NULL,
    `order_date` DATETIME NOT NULL,
    `total_amount` DECIMAL(8, 2) NOT NULL,
    `payment_method` INT NOT NULL,
    `shipping_address` INT UNSIGNED NOT NULL,
    `order_status` INT NOT NULL DEFAULT '1'
);
CREATE TABLE `OrderDetails`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `order_id` INT UNSIGNED NOT NULL,
    `book_id` INT UNSIGNED NOT NULL,
    `quantity` INT UNSIGNED NOT NULL,
    `unit_price` DECIMAL(8, 2) NOT NULL
);
CREATE TABLE `orderstatus`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `status_name` VARCHAR(10) NOT NULL
);
CREATE TABLE `paymentmethod`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `method_name` VARCHAR(15) NOT NULL
);
```

Figure 1.6 (Showing SQL DDL queries to the tables responsible for payment process)

In Figure 1.7, we can see the relationships between the 'Customer' table and all the necessary tables for conducting operations related to customers in the system. I would like to emphasize the 'credit card' table here. We established a foreign key relationship by writing its IDs into the 'Customer' table.

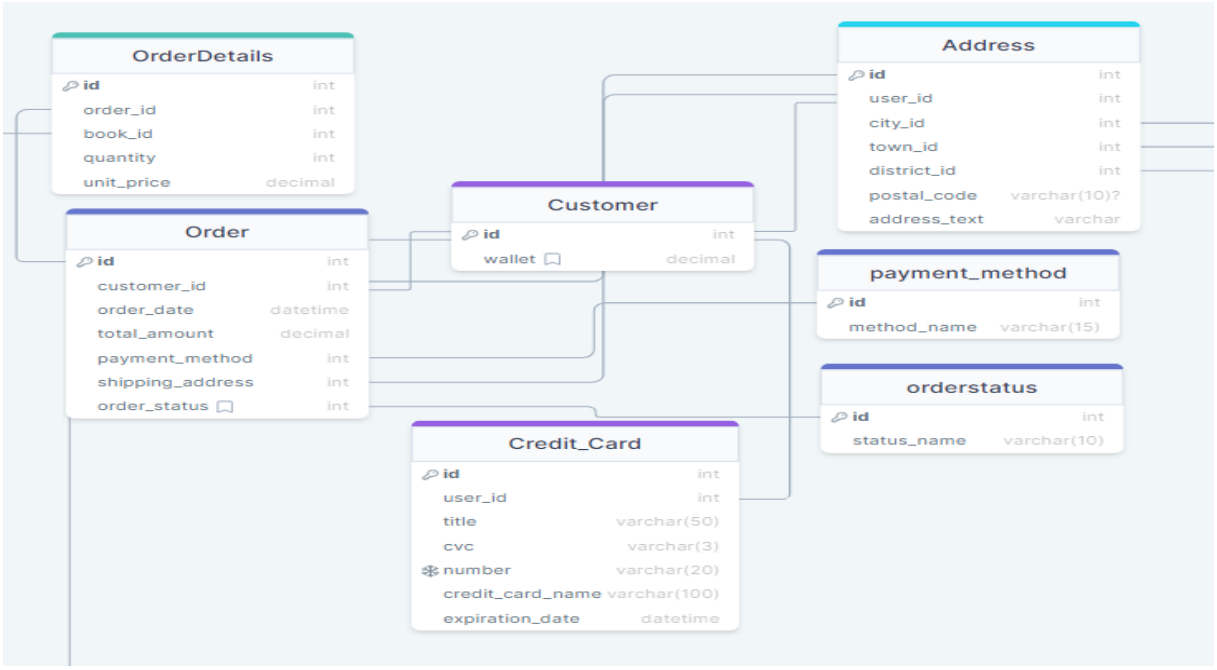


Figure 1.7 (displaying all the tables associated with our Customer table.)

In Figure 1.8, we see standard DDL configurations. We use a unique key constraint for the 'credit card number' table since it is a distinguishing feature for credit card data."

```sql
CREATE TABLE `Customer`(
    `id` INT UNSIGNED NOT NULL PRIMARY KEY,
    `wallet` DECIMAL(8, 2) NOT NULL DEFAULT '0'
);
CREATE TABLE `Credit_Card`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `user_id` INT NOT NULL,
    `title` VARCHAR(50) NOT NULL,
    `cvc` VARCHAR(3) NOT NULL,
    `number` VARCHAR(20) NOT NULL,
    `credit_card_name` VARCHAR(100) NOT NULL,
    `expiration_date` DATETIME NOT NULL
);
ALTER TABLE
    `Credit_Card` ADD UNIQUE `credit_card_number_unique`(`number`);
```

Figure 1.8 (Showing SQL DDL queries to form costumer and credit card tables)

As seen in Figure 1.9, instead of writing address values directly into the address table, we create a more cost-effective system in terms of memory management and resource usage by writing the ID values of foreign key tables."

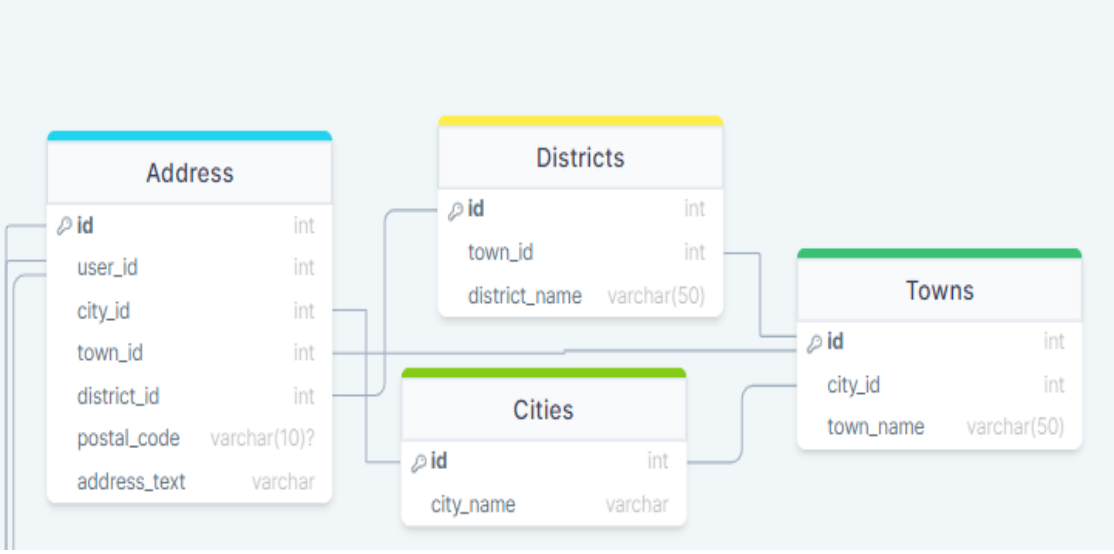


Figure 1.9 (showing the address table and the foreign key tables associated with the address table.)

In Figure 1.10, we define the relevant fields in the address table to hold foreign key references to other tables. Additionally, we establish relationships by specifying 'city_id' for 'town' and 'town_id' for 'district,' as seen in Figure 1.9.

```sql
CREATE TABLE `Address`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `user_id` INT UNSIGNED NOT NULL,
    `city_id` INT UNSIGNED NOT NULL,
    `town_id` INT UNSIGNED NOT NULL,
    `district_id` INT UNSIGNED NOT NULL,
    `postal_code` VARCHAR(10) NULL,
    `address_text` VARCHAR(255) NOT NULL
);
CREATE TABLE `Cities`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `city_name` VARCHAR(255) NOT NULL
);
CREATE TABLE `Towns`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `city_id` INT UNSIGNED NOT NULL,
    `town_name` VARCHAR(50) NOT NULL
);
CREATE TABLE `Districts`(
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `town_id` INT UNSIGNED NOT NULL,
    `district_name` VARCHAR(50) NOT NULL
);
```

Figure 1.10 (Showing SQL DDL queries to form address and its foreign key tables)

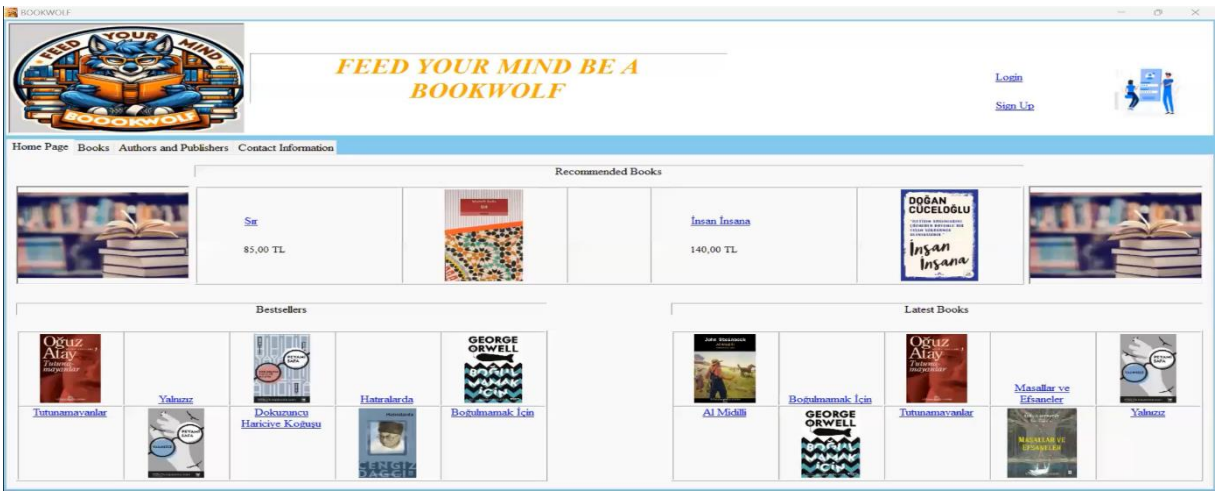
System Modules and Functional Descriptions

Frontend:

```
MySqlConnection connection = new MySqlConnection("Data Source = localhost; Initial Catalog = databaseApp; User ID = root; Password = 0000"); // database connection
```

Figure 1 (Database Connection)

I defined the relevant MySQL connection to the application as shown in figure 1.



1-) FORM1 (Home Page Screen):

Figure 2 (Application Screen)

When we run the application, we first encounter the screen in figure 2. There is a linklabel object in the upper menu for the login and signup process. After the login or signup process, some user information appears on the upper menu, as shown in figure 3.

Figure 3 (Upper Menu)

```
"SELECT u.id, u.username, u.first_name, u.last_name, c.wallet, u.is_manager " +
"FROM users u " +
"LEFT JOIN customer c ON u.id = c.id " +
"WHERE u.id = @userID", connection);
```

In figure 3, I used the SQL query in figure 4 from the user and customer tables to print the required user information in the top menu.

Figure 4 (Upper Menu SQL Command)

```
("SELECT id, title, price, image FROM Book ORDER BY RAND() LIMIT 1"
timer1 = new Timer();
timer1.Interval = 3000;
timer1.Tick += Timer_Tick;
timer1.Start();
```

Using the SQL command in figure 5 from the book table in figure 2, I brought random book information to the home page screen with the help of the timer object.

Figure 5 (Recommended Books SQL Command)

```
("SELECT id, title, image FROM book ORDER BY number_of_sales DESC LIMIT 5"
```

I used the SQL command in figure 6 from the book table to get the information of the five best-selling books on the home page screen in figure 2.

Figure 6 (Bestsellers Book SQL Command)

```
("SELECT id, title, image FROM book ORDER BY publication_date DESC LIMIT 5"
```

Figure 7 (Latest Book SQL Command)

I used the SQL command in figure 7 from the book table to get the information of the five latest books on the home page screen in figure 2.

```
linkLabelArray[i].Text = dataTable.Rows[i]["title"].ToString();
linkLabelArray[i].Tag = Convert.ToInt32(dataTable.Rows[i]["id"]); // Used to go to book page

byte[] imageBytes = dataTable.Rows[i]["image"] as byte[]; // To take a picture from a sql table into a picture box

if (imageBytes != null && imageBytes.Length > 0)
{
    try
    {
        using (MemoryStream ms = new MemoryStream(imageBytes))
        {
            pictureBoxList[i].Image = System.Drawing.Image.FromStream(ms); // state of image
            pictureBoxList[i].SizeMode = PictureBoxSizeMode.Zoom;
            pictureBoxList[i].Tag = Convert.ToInt32(dataTable.Rows[i]["id"]); // Used to go to book page
        }
    }
```

As seen in figure 8, I added a link to the linklabel and picturebox objects in figure 2 to navigate to the book page.

Figure 8 (add link)

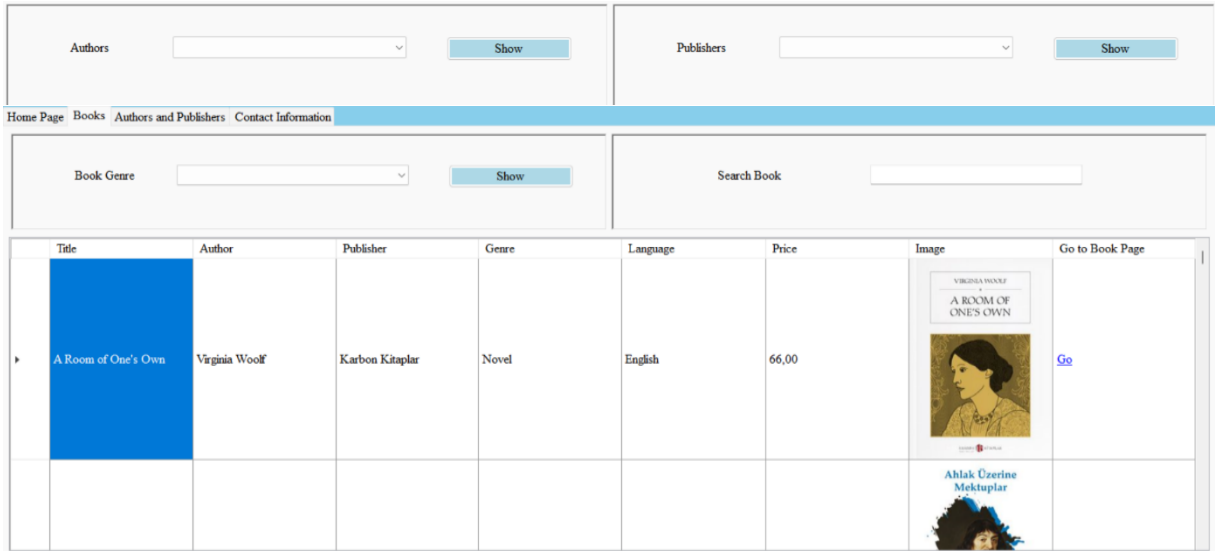


Figure 9 (Books, Publisher, and Author Page)

```
"SELECT b.id, b.title AS 'Title', " +
"a.author_name AS 'Author', " +
"p.publisher_name AS 'Publisher', " +
"g.genre_name AS 'Genre', " +
"l.language_name AS 'Language', " +
"b.price AS 'Price', b.image AS 'Image' " +
"FROM Book b " +
"INNER JOIN Author a ON b.author = a.id " +
"INNER JOIN Publisher p ON b.publisher = p.id " +
"INNER JOIN Genre g ON b.genre = g.id " +
"INNER JOIN Language l ON b.language = l.id " +
"WHERE g.id = @selectedGenreId " +
"ORDER BY b.title", connection);
```

```
"SELECT b.id, b.title AS 'Title', a.author_name AS 'Author', " +
"p.publisher_name AS 'Publisher', g.genre_name AS 'Genre', " +
"l.language_name AS 'Language', b.price AS 'Price', " +
"b.image AS 'Image' " +
"FROM Book b " +
"INNER JOIN Author a ON b.author = a.id " +
"INNER JOIN Publisher p ON b.publisher = p.id " +
"INNER JOIN Genre g ON b.genre = g.id " +
"INNER JOIN Language l ON b.language = l.id " +
"WHERE b.title LIKE '" + textBox1.Text + "%'" +
"ORDER BY b.title", connection);
```

Figure 10 (Search Book SQL Commands)

You can search by the book genre, title, author, and publishing house you want on the page screen in figure 9. While making these calls, I used the SQL commands in figure 10 that call the book, author, publisher, genre, language tables to print to the datagridview object.

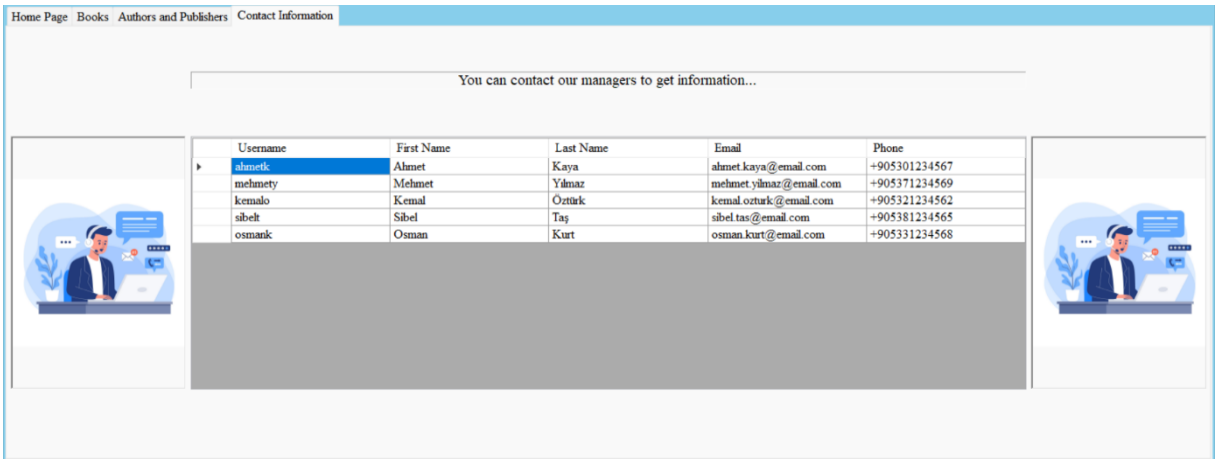


Figure 11 (Manager Page)

```
"SELECT username AS 'Username', first_name AS 'First Name', " +
"last_name AS 'Last Name', email AS 'Email', phone AS 'Phone'" +
"FROM users WHERE is_manager = 1", connection);
```

Figure 12 (see Manager SQL Command)

We brought the manager information as seen in figure 11. In this way, people will be able to contact us. While doing this, I used the SQL command in figure 12, which calls the users table.

2-) FORM2 (Book Screen):

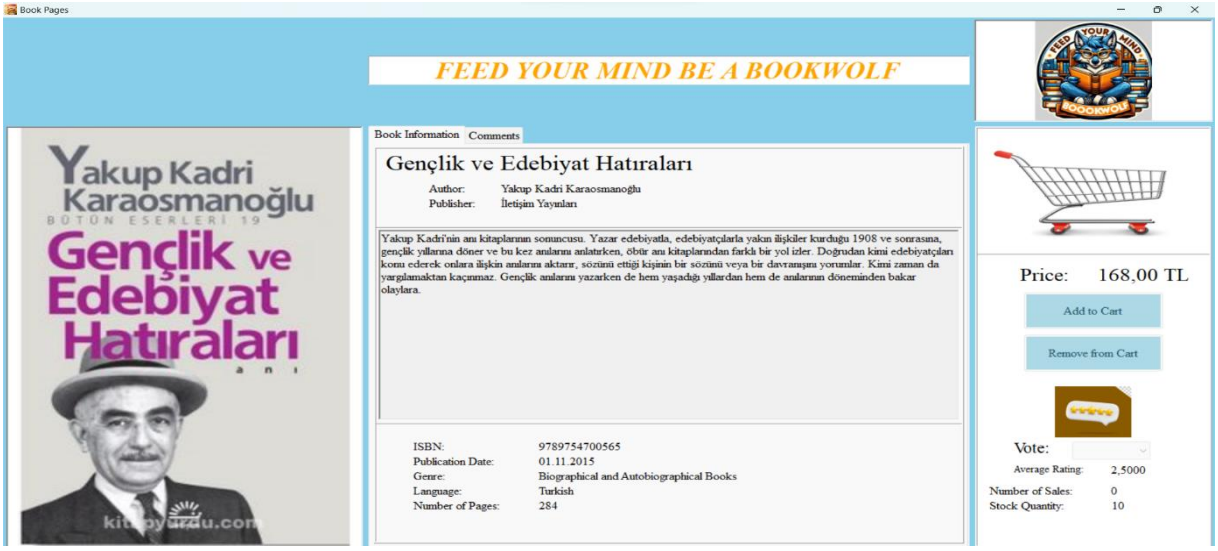


Figure 13 (Book Screen)

You can access this page by clicking on the pictures on the home page, the name of the book, or the go button on the search screen. When there is no user login on this page, the buttons are inactive.

```
"SELECT b.title, a.author_name, b.isbn, b.price, " +
"b.stock_quantity, p.publisher_name, b.publication_date, " +
"b.number_of_sales, b.number_of_page, b.description, " +
"g.genre_name, l.language_name, b.image " +
"FROM book b " +
"INNER JOIN Author a ON b.author = a.id " +
"INNER JOIN Publisher p ON b.publisher = p.id " +
"INNER JOIN Genre g ON b.genre = g.id " +
"INNER JOIN Language l ON b.language = l.id " +
"WHERE b.id = @bookID", connection);
```

To display the book information on the screen in figure 13, I used the SQL command that calls the book, author, publisher, genre, language tables in figure 14.

Figure 14 (see Book Information)

```
"stock_quantity"]) > 0 && cart != null)

"UPDATE book SET number_of_sales = number_of_sales + 1, " +
"stock_quantity = stock_quantity - 1 WHERE id = @BookID", co
"UPDATE book SET number_of_sales = number_of_sales - 1, " +
"stock_quantity = stock_quantity + 1 WHERE id = @BookID", co
```

When you want to add or remove products from the cart, as in figure 13, the command that updates the book table in figure 15 is called. If the book is in stock, it adds it into the basket.

Figure 15 (add or remove Cart SQL Command)

```
"SELECT book_id, AVG(rating) AS average_rating " +
"FROM rating WHERE book_id = @bookID GROUP BY book_id",
if (count > 0)
{ // update rating
    addUpdateRating("UPDATE rating SET rating = @rating WHERE book_id = @bookID AND user_id = @userID", selectedRating);
}
else
{ // add rating
    addUpdateRating("INSERT INTO rating (book_id, user_id, rating) VALUES (@bookID, @userID, @rating)", selectedRating);
}
```

In figure 13, there is a combobox for voting. Here, I called the SQL command that calls the rating table in figure 16 to see the average rating and vote.

Figure 16 (Vote SQL Commands)

Figure 17 (Comment Page)

```
"SELECT c.id AS 'id', u.username AS 'Username', " +
"u.first_name AS 'First Name', " +
"u.last_name AS 'Last Name', c.comment AS 'Comment' " +
"FROM comment c " +
"INNER JOIN users u ON u.id = c.user_id " +
"WHERE c.book_id = @bookID", connection);

"INSERT INTO comment(book_id, user_id, comment) " +
"VALUES(@bookID, @userID, @commentText)", connection);

"DELETE FROM comment WHERE id = @commentID"
```

It is possible to add and delete comments in Figure 17. To delete a comment, my comment button must be clicked. For this, I used the SQL command in figure 18, which calls insert, comment for delete, and the combined user table.

Figure 18 (Comment SQL Commands)

3-) FORM3 (Login Screen):

In this section, we have separate login screens for admins and users. We send requests to the API, and if the user is registered in the system, we receive a token for session management. This token is essential for maintaining secure and authenticated sessions, ensuring that each user's experience is both safe and personalized.

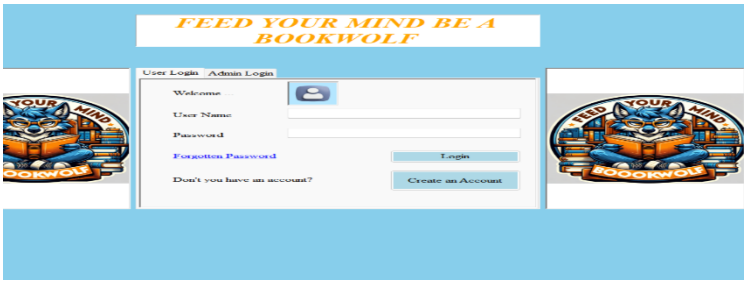


Figure 41 (Login Screen)

Here are the SQL codes we use for the login process.

```
"SELECT * FROM Users WHERE username = ?",
'UPDATE Users SET last_login_date = ? WHERE id = ?", [lastLoginDate, userId]
```

Figure 42 (SQL Login Commands

4-) FORM4 (Sign Up Screen):

In this section, we handle the process of users signing up to the system. We collect the information entered by the users and send it to the API. This is a crucial step for user registration, where we ensure that all necessary data is correctly gathered and securely transmitted to the backend for further processing and user account creation.

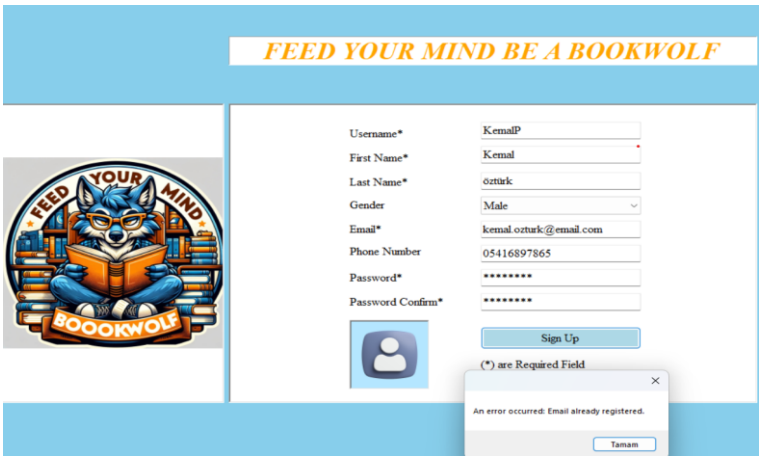


Figure 43 (Signup Screen)

```
("INSERT INTO Users SET ?", newUser,
"INSERT INTO customer SET ?", customer,
```

Here, we see the SQL code that writes the user data to the database during the signup process.

Figure 44 (Login SQL Commands)

5-) FORM5 (Cart Screen):

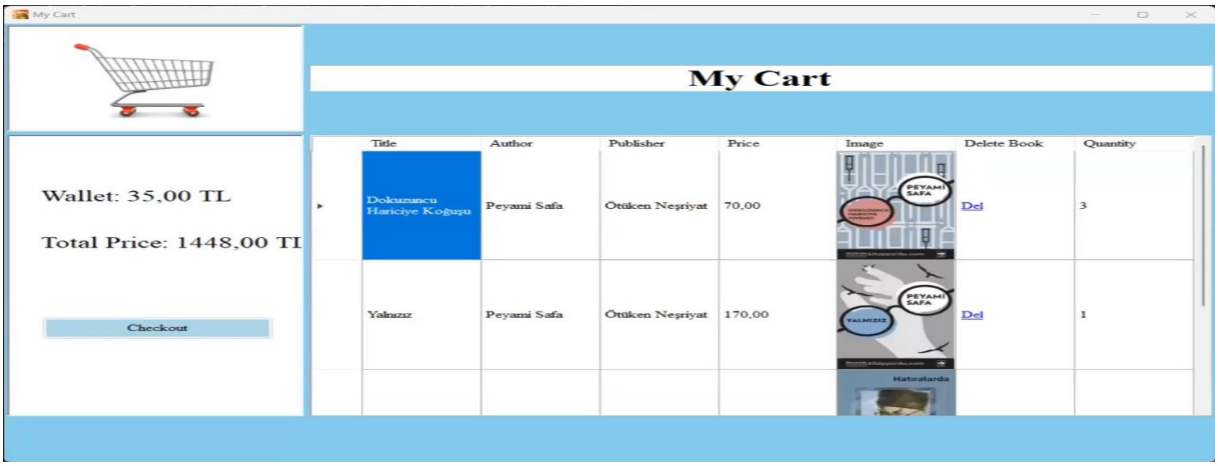


Figure 19 (Cart Screen)

```
"SELECT b.id, b.title AS 'Title', a.author_name AS 'Author', " +
"p.publisher_name AS 'Publisher', b.price AS 'Price', " +
"b.image AS 'Image' " +
"FROM Book b " +
"INNER JOIN Author a ON b.author = a.id " +
"INNER JOIN Publisher p ON b.publisher = p.id " +
$"WHERE b.id IN ({string.Join(",", cart)})", connection);
```

```
"SELECT u.id, c.wallet " +
"FROM users u " +
"LEFT JOIN customer c ON u.id = c.id " +
"WHERE u.id = @userID", connection);
```

As seen in figure 19, the products in the basket can be viewed. If you wish, you can remove the book you choose from the basket. When the checkout button is pressed, it sends you to the Complete Shopping screen. For cart information, I used the book, author, publisher, user, customer tables in figure 20.

Figure 20 (Cart SQL Commands)

6-) FORM6 (Customer Account Settings Screen):

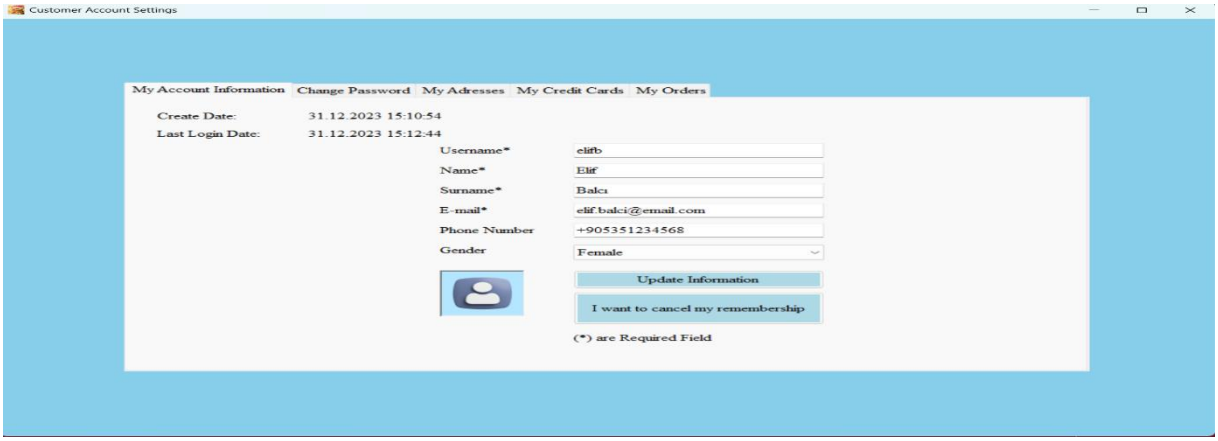


Figure 21 (Customer Account Settings Screen)

On the screen in figure 21, the customer can make requests such as checking, deleting, and updating relevant information.

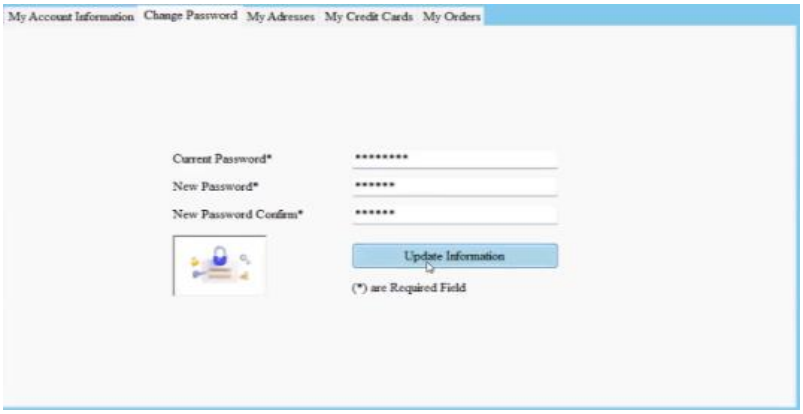
```
"SELECT username, first_name, last_name, " +
"gender, email, phone, create_date, last_login_date, c.wallet " +
"FROM users " +
"LEFT JOIN customer c ON c.id = users.id " +
"WHERE users.id = @userID", connection);
"UPDATE users " +
"SET username = @newUsername, " +
"first_name = @newFirst_name, " +
"last_name = @newLast_name, " +
"email = @newEmail, " +
"phone = @newPhone, " +
"gender = @newGender " +
"WHERE id = @userID", connection);
```

```
"PATCH"), $"api/v1/users/{userID}"
```

```
"SELECT 1 FROM address WHERE user_id = @userID", connec
alue("@userID", userID);
and.ExecuteScalar();

and("DELETE FROM address WHERE user_id = @userID", conn
ithValue("@userID", userID);

"SELECT 1 FROM credit_card WHERE user_id = @userID", co
alue("@userID", userID);
ommand.ExecuteScalar();

1)

and("DELETE FROM credit_card WHERE user_id = @userID",
ithValue("@userID", userID);

"SELECT 1 FROM `order` WHERE customer_id = @userID", co
alue("@userID", userID);
d.ExecuteScalar();

and("DELETE FROM `order` WHERE customer_id = @userID",
ithValue("@userID", userID);

"DELETE FROM users WHERE id = @userID", connection);
```

Figure 22 (Customer Information SQL Commands)

In figure 21, there is a button to update and delete user information. For this purpose, users, customer, tables are called. The required SQL codes are given in figure 22. Additionally, if you want to perform a delete operation, it deletes the user from all tables, as seen in figure 22. Additionally, a patch is sent to the usersAPIforthese operations. During the update process, field validations are performed through the API. This ensures that all data being updated meets the system's requirements and standards for accuracy and integrity.

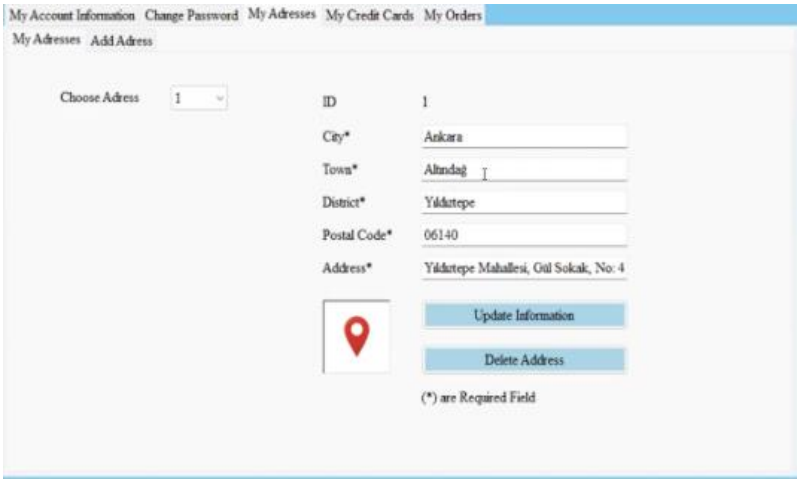


As seen in figure 23, there is an option to change the password. In figure 24, this process is done by calling the users table and sending a patch to the API.

Figure 23 (Change Password Page)

```
("PATCH"), "api/v1/users/updatePass")
"UPDATE users SET password_hash = @newPassword WHERE id = @userID",
```

Figure 24 (Update Password SQL Command)



On the address screen in figure 25, the user can view, update, delete or add new address information. For this, I used the necessary SQL commands for the address, cities, towns, districts, tables as shown in figure 26.

Figure 25 (Address Page)

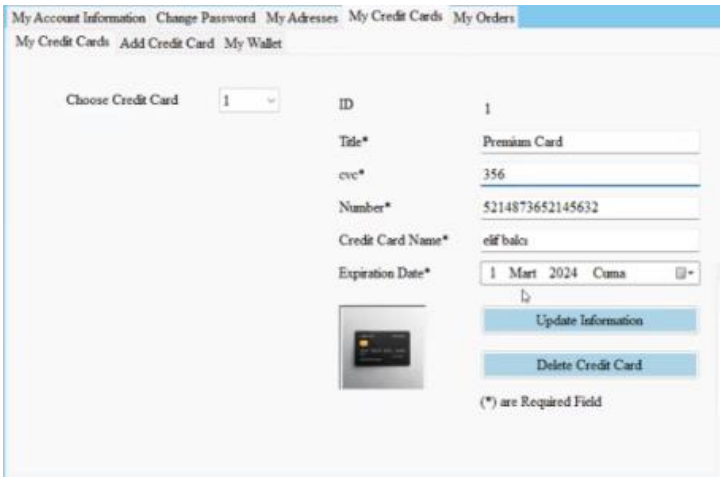
```
"SELECT a.id, c.city_name AS 'City', t.town_name AS 'Town', " +
"d.district_name AS 'District', a.postal_code AS 'Postal', " +
"a.address_text AS 'AddressText' " +
"FROM address a " +
"LEFT JOIN cities c ON c.id = a.city_id " +
"LEFT JOIN towns t ON t.id = a.town_id " +
"LEFT JOIN districts d ON d.id = a.district_id " +
"WHERE a.user_id = @userID LIMIT 1 OFFSET @addressID", connection);
```

```
"INSERT INTO Address (user_id, city_id, town_id, district_id, postal_code, address_text) " +
"VALUES (@userID, @city, @town, @district, @postalCode, @addressText)", connection);
```

```
"UPDATE Address " +
"SET city_id = @city, " +
"town_id = @town, " +
"district_id = @district, " +
"postal_code = @postalCode, " +
"address_text = @addressText " +
"WHERE id = @addressID", connection);
```

```
"DELETE FROM address WHERE id = @addressID",
```

Figure 26 (Change Address SQL Commands)



On the credit card screen in Figure 27, the user can view, update, delete or add new credit card information. I used SQL commands to call the Credit_card table required for this, as seen in Figure 28.

Figure 27 (Credit Card Page)

```
"SELECT id, title, cvc, number, credit_card_name, expiration_date " +
"FROM credit_card WHERE user_id = @userID LIMIT 1 OFFSET @cardID", co
```

```
"INSERT INTO credit_card (user_id, title, cvc, number, credit_card_name, expiration_date) " +
"VALUES (@userID, @title, @cvc, @number, @credit_card_name, @expiration_date)", connection);
```

```
"DELETE FROM credit_card WHERE id = @cardID"
```

```
"UPDATE credit_card " +
"SET title = @title, " +
"cvc = @cvc, " +
"number = @number, " +
"credit_card_name = @credit_card_name, " +
"expiration_date = @expiration_date " +
"WHERE id = @cardID ", connection);
```

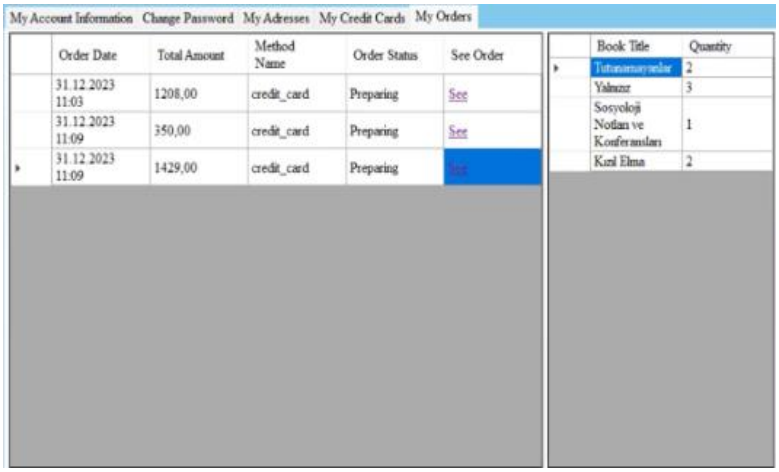
Figure 28 (Credit Card SQL Command)

In figure 29, it is possible to load money to the wallet in my wallet tab. The SQL codes required to call the customer table are as shown in figure 30.

Figure 29 (add Money)

```
"UPDATE customer " +
"SET wallet = @wallet "
"WHERE id = @userID", co
```

Figure 30 (update Wallet SQL Command)



We used the SQL commands in figure 32, which call the order, orderdetails, paymentmethod, orderstatus tables, to display the user's past orders and the book information on the order he/she will select, as shown in figure 31.

Figure 31 (Order Page)

```
"SELECT o.id AS 'id', o.order_date AS 'Order Date', " +
"o.total_amount AS 'Total Amount', p.method_name AS 'Method Name', " +
"s.status_name AS 'Order Status' " +
"FROM `order` o " +
"LEFT JOIN paymentmethod p ON p.id = o.payment_method " +
"LEFT JOIN orderstatus s ON s.id = o.order_status " +
"WHERE o.customer_id = @userID", connection);
```

```
"SELECT b.id, b.title AS 'Book Title', o.quantity AS 'Quantity' " +
"FROM book b " +
"LEFT JOIN orderdetails o ON b.id = o.book_id " +
"WHERE o.order_id = @orderID", connection);
```

Figure 32 (Select order SQL Commands)

7-) FORM7 (Manager Account Settings Screen):

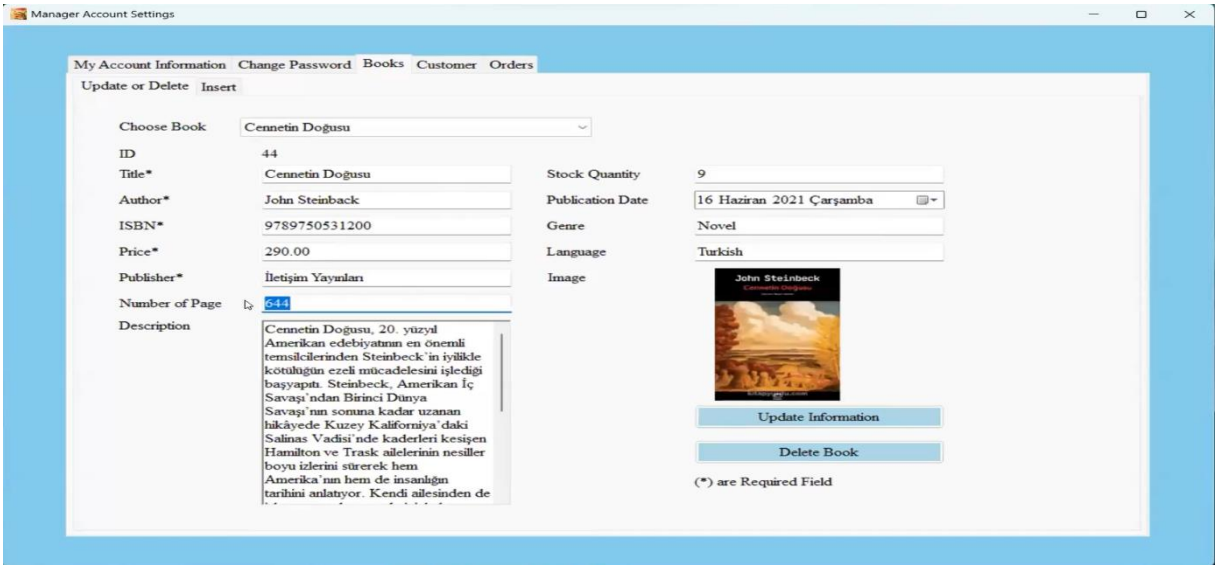


Figure 33 (Manager Account Settings Screen)

The password renewal and account information pages on the manager account settings screen in figure 23 are the same as in figure 21.

```
"SELECT b.id, b.title, a.author_name, b.isbn, b.price, " +
"b.stock_quantity, p.publisher_name, b.publication_date, " +
"b.number_of_page, b.description, g.genre_name, " +
"l.language_name, b.image " +
"FROM book b " +
"INNER JOIN Author a ON b.author = a.id " +
"INNER JOIN Publisher p ON b.publisher = p.id " +
"INNER JOIN Genre g ON b.genre = g.id " +
"INNER JOIN Language l ON b.language = l.id " +
"WHERE b.id = @bookID", connection);
```

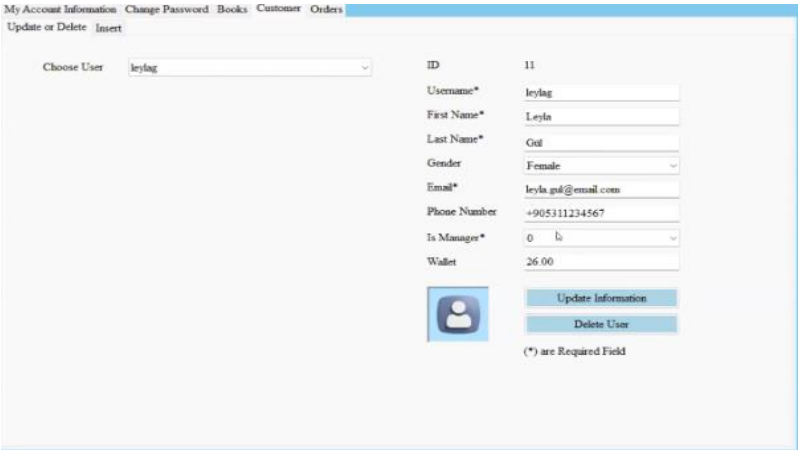
```
"DELETE FROM book WHERE id = @bookId", connec
alue("@bookId", label44.Text);

rID, null);

"DELETE FROM rating WHERE book_id = @bookId",

alue("@bookId", label44.Text);

"DELETE FROM comment WHERE book_id = @bookId"
```

```
"UPDATE Book " +
"SET " +
"title = @title, " +
"author = @authorID, " +
"isbn = @isbn, " +
"price = @price, " +
"stock_quantity = @stock_quantity, " +
"publisher = @publisherID, " +
"publication_date = @publication_date, " +
"number_of_page = @number_of_page, " +
"description = @description, " +
"genre = @genreID, " +
"language = @languageID " +
"WHERE id = @bookID", connection);
```

```
"INSERT INTO Book " +
"(title, author, isbn, price, stock_quantity, publisher, publication_date, number_of_page, description, genre, language) " +
"VALUES " +
"(@title, @authorID, @isbn, @price, @stock_quantity, @publisherID, @publication_date, @number_of_page, @description, @genreID, @languageID)",
```

In figure 33, it is possible to review, update, delete or add new book information. For this, we used SQL commands that call the book, author, publisher, genre, language, rating, comment tables in Figure 34. In figure 33, it is possible to review, update, delete or add new book information. For this, we used SQL commands that call the book, author, publisher, genre, language, rating, comment tables in Figure 34. In this part, we also utilize various validations in the create and update sections that are implemented in the API. For example, one of the validations ensures that the ISBN is unique. This is crucial to maintain data integrity and prevent duplication in the system.

Figure 34 (Book Information SQL Commands)



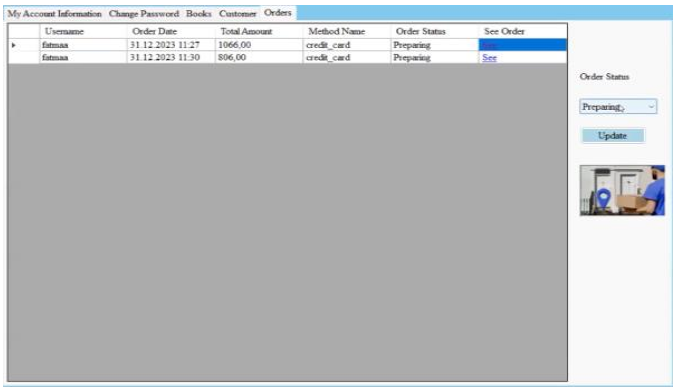
In figure 35, it is possible to review, update, delete user information and add new users. For this, I used the SQL commands in figure 36, which call the users, customer tables. I also did this by sending a patch to the API. In this area, for the update process through the API, we have written detailed validations for the fields. This includes checks for unique fields like username and email, as well as validation for the format of fields.

Figure 35 (Customer Page)

```
"SELECT u.id, u.username, u.first_name, " +
"u.last_name, u.gender, u.email, u.phone, " +
"u.is_manager, u.password_hash, c.wallet " +
"FROM users u " +
"LEFT JOIN customer c ON c.id = u.id " +
"OR c.id IS NULL " +
"WHERE u.id = @userID", connection);
"DELETE FROM users WHERE id = @userID",

"INSERT INTO users (username, first_name, last_name, " +
"gender, email, phone, is_manager, password_hash, create_date) " +
"VALUES " +
"(@username, @firstName, @lastName, @gender, @email, @phone, " +
"@isManager, @passwordHash, NOW()); " +
"INSERT INTO customer (id, wallet) " +
"VALUES " +
"(LAST_INSERT_ID(), @wallet)", connection);
```

```
"UPDATE users u " +
"INNER JOIN customer c ON c.id = u.id " +
"SET " +
"u.username = @username, " +
"u.first_name = @firstName, " +
"u.last_name = @lastName, " +
"u.gender = @gender, " +
"u.email = @email, " +
"u.phone = @phone, " +
"u.is_manager = @isManager, " +
"u.password_hash = @passwordHash, " +
"c.wallet = @wallet " +
"WHERE u.id = @userID", connection);

"PATCH"), $"api/v1/users/{userID}"
```

Figure 36 (Customer SQL Commands)



In practice, managers can edit the order status as shown in Figure 37. The SQL commands that perform this operation and call the order and orderstatus tables are as shown in Figure 38.

Figure 37 (Set Order Page)

```
"SELECT s.status_name FROM `order` o " +
"LEFT JOIN orderstatus s ON s.id = o.order_status " +
"WHERE o.id = @orderID", connection);
"UPDATE `order` SET order_status = @order_status WHERE id = @orderID",
```

Figure 38 (Set Order SQL Commands)

8-) FORM8 (Reset Password Screen):

In this section, we manage our password reset process by sending a request to our API's mail server. Our mail service then sends us a reset token, which is valid for 10 minutes, to facilitate secure password resetting.

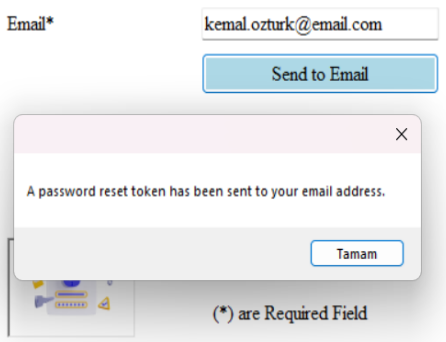


Figure 44 (Reset Password Screen)

Here, we see the SQL codes.

```
"SELECT * FROM Users WHERE email = ?", [email],
"UPDATE Users SET reset_Token = ?, reset_Token_Expires = ? WHERE email = ?", [resetTokenHash, resetTokenExpires, email],
```

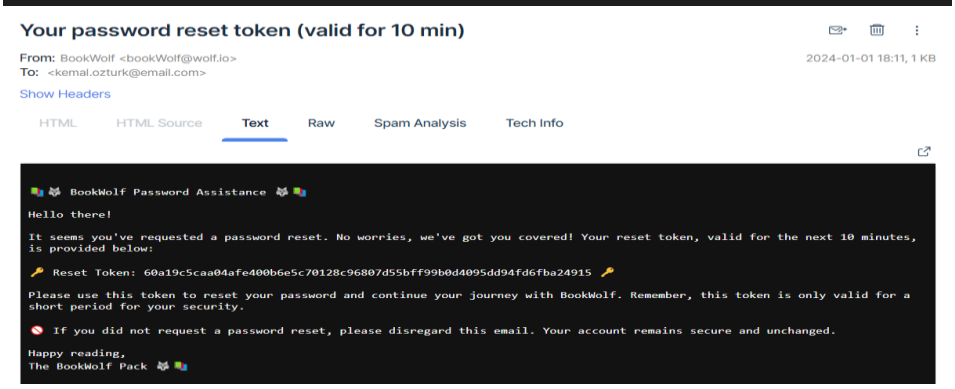


Figure 45 (Reset Password Token and SQL Commands)

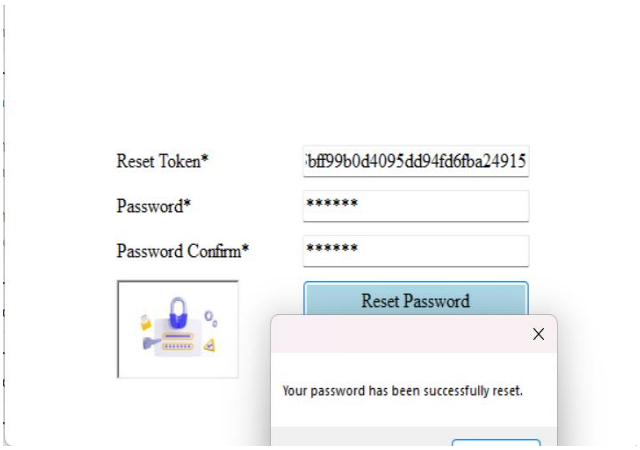


Figure 46 (Reset Password Token)

Here, we see the SQL codes.

```
'SELECT * FROM Users WHERE reset_Token = ?", [hashedToken],
"UPDATE Users SET password_hash = ?, reset_Token = NULL, reset_Token_Expires = NULL WHERE id = ?", [newPassword, userId],
```

Figure 47 (Reset Token SQL Commands)

9-) FORM9 (Complete Order Screen):

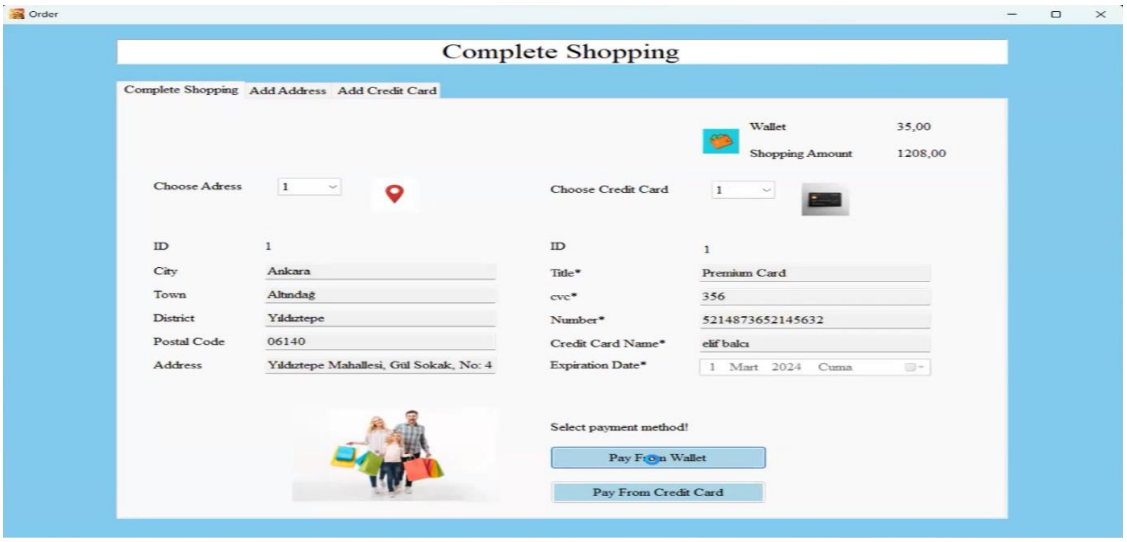


Figure 39 (Complete Shopping Screen)

```
"SELECT a.id, c.city_name AS 'City', t.town_name AS 'Town', " +
"d.district_name AS 'District', a.postal_code AS 'Postal', " +
"a.address_text AS 'AddressText' " +
"FROM address a " +
"LEFT JOIN cities c ON c.id = a.city_id " +
"LEFT JOIN towns t ON t.id = a.town_id " +
"LEFT JOIN districts d ON d.id = a.district_id " +
"WHERE a.user_id = @userID LIMIT 1 OFFSET @addressID", connection);
```

```
"SELECT id, title, cvc, number, credit_card_name, expiration_date " +
"FROM credit_card WHERE user_id = @userID LIMIT 1 OFFSET @cardID", co
```

```
"INSERT INTO orderdetails (order_id, book_id, quantity, unit_price) " +
"VALUES (@order_id, @book_id, @quantity, @unit_price)", connection);
```

```
"INSERT INTO `order` (customer_id, order_date, total_amount, payment_method, shipping_address) " +
"VALUES (@customer_id, NOW(), @total_amount, @payment_method, @shipping_address)", connection);
```

It is possible to complete the order on the screen in Figure 39. After selecting an address, if you choose a credit card or if your balance is sufficient, you can complete the shopping from the wallet. SQL codes that call the address, cities, district, towns, credit card, order, orderdetails tables for the necessary transactions are available in figure 40. The opportunity to add an address or a credit card is also available on this page. For this, I used the SQL commands in figure 26 and figure 28.

Figure 40 (Shopping SQL Commands)

Backend

In this section, we will have several code snippets from our backend, accompanied by additional explanations. We will discuss the improvements we have made related to authentication processes, security, and field validations.

User Registration and Password Security

In this section, we see the code snippets where we perform the hashing of passwords and generate unique JWT (JSON Web Tokens) for each user. These processes are important because they ensure a high level of security and integrity within our system.

Hashing passwords is crucial for protecting user credentials. By converting passwords into hashed forms, we prevent the actual password values from being stored or transmitted in a readable format, thus safeguarding them against unauthorized access and potential breaches.

Generating unique JWTs for each user is equally important for maintaining secure and authenticated sessions. JWTs are used to verify the identity of users after they log in, allowing for secure communication between the client and the server. This ensures that each user's interaction with the system is secure, personalized, and consistent with their access rights.

Overall, these security measures are vital for maintaining user trust and protecting sensitive data within our application.

```
function updateAllUsers() {
    connection.query('SELECT id, password_hash FROM users', (err, users) => {
        if (err) {
            console.error("Error while fetching users:", err);
            return;
        }

        users.forEach(user => {
            // The current plain text passwords of the users
            const currentPlainTextPassword = user.password_hash;

            bcrypt.hash(currentPlainTextPassword, saltRounds, (err, hash) => {
                if (err) {
                    console.error("Hashing error:", err);
                    return;
                }

                const token = jwt.sign({ id: user.id }, jwtSecret, {
                    expiresIn: '90d'
                });

                updateUserPasswordAndToken(user.id, hash, token);
            });
        });
    });
}
```

phone	password_hash	create_date	last
+905301234567	pass1234	2024-01-01 18:18:37	202
+905351234568	1234pass	2024-01-01 18:18:37	202
+905371234569	abcd1234	2024-01-01 18:18:37	202
+905391234560	password1	2024-01-01 18:18:37	202
+905301234561	mypassword	2024-01-01 18:18:37	202
+905321234562	pass9876	2024-01-01 18:18:37	202
+905341234563	simplepass	2024-01-01 18:18:37	202
+905361234564	qwerty123	2024-01-01 18:18:37	202

phone	password_hash	create_date
+905301234567	$2b$10$9Pe8pa1JZCs5eCei6jkdUem/MbMgW2P...	2024-01-01 18:18:37
+905351234568	$2b$10$ucJwOncGQGXABDyoejVU3elfl14NQjW...	2024-01-01 18:18:37
+905371234569	$2b$10$pnlUm8WW/gp2.XumRRWyx.rEjQIL3M...	2024-01-01 18:18:37
+905391234560	$2b$10$ZM8vnDyruiFF51xnAUy49elVT1qjrwOe...	2024-01-01 18:18:37
+905301234561	$2b$10$yg/owypa8p58hyU3anOuiekIKnMSciSh...	2024-01-01 18:18:37
+905321234562	$2b$10$WnT/t3IFADaoA03oksFOrOb6G6yi1EX...	2024-01-01 18:18:37
+905341234563	$2b$10$F8BaZfIzES3afqhbhFMJBufSti849/1ab5...	2024-01-01 18:18:37
+905361234564	$2b$10$vY3vh1BYLNLObJJLoJZaUex1aCY1lavo...	2024-01-01 18:18:37

Figure 48 (Password Hash)

JWT for Authentication

In these code snippets below, we can observe the implementation of JWT (JSON Web Tokens) in the signup process. This integration is essential as it adds a layer of security and ensures a seamless user experience. During signup, once a user's credentials are verified and

their account is created, a JWT is generated. This token serves as a secure means of identifying the user in subsequent requests to the server. It encapsulates the user's identity and is encrypted, making it a safe way to maintain user sessions without constantly requiring username and password verification. The use of JWT in the signup process not only enhances security but also improves the efficiency of the system. It allows for stateless authentication, meaning the server doesn't need to store session information about the user. Instead, it can validate the user's identity by decrypting the token, making the system more scalable and responsive. In summary, the inclusion of JWTs in the signup process is a crucial aspect of our backend implementation, ensuring secure, efficient, and user-friendly authentication.

```
        // Generate a token for the new user
        const token = jwt.sign({ id: data.id }, process.env.JWT_SECRET, {
            expiresIn: process.env.JWT_EXPIRES_IN
        });

        // Send a success response
        res.status(201).send({
            message: "User successfully created.",
            id: data.id,
            token // Send the token to the user
        });
    });
} catch (error) {
    // Handle any errors that occur during user object creation
```

```
exports.signUp = (req, res) => {
    // Extract user details from request body
    const { username, first_name, last_name, gender, email, phone, password, passwordConfirm } = req.body;

    // Password validation
    if (password !== passwordConfirm) {
        return res.status(400).send({ message: "Passwords do not match." });
    }
    // Hash the password
    bcrypt.hash(password, saltRounds, (err, hash) => {
        if (err) {
            return res.status(500).send({ message: "Error hashing password." });
        }
        try {
            // Create a new user object
            const newUser = new User({
                username,
                first_name,
                last_name,
                gender,
                email,
                phone,
                password_hash: hash
            });
            // Add the user to the database
            User.createUser(newUser, (error, data) => {
                if (error) {
                    if (error.kind === "username_exists") {
                        return res.status(409).send({ message: "Username already exists." });
                    } else if (error.kind === "email_exists") {
                        return res.status(409).send({ message: "Email already registered." });
                    } else {
                        return res.status(500).send({ message: error.message });
                    }
                }
```


```
"message": "User successfully created.",
"id": 22,
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJpZCI6MjIsImlhdCI6MTcwNDA0ODA4MiwiZXhwIjoxNzExODI0MDgyfQ.
    ZkfB7baTPbFwaF0fimrWwpbB6SKMKdLLNn3xnK7l4Ns"
```


Figure 49 (Backend Code)

Restriction Of Access

In this section, we see code snippets where we have restricted external access to our API endpoints using protection mechanisms. This process is important because it ensures that only authenticated and authorized users can interact with our system, thereby safeguarding sensitive data and functionalities. By implementing access control on our API endpoints, we prevent unauthorized or malicious requests from accessing or manipulating data. This is typically achieved using authentication tokens like JWTs, which validate a user's identity before granting them access to certain endpoints. Only requests with valid tokens, which indicate that the user is logged in and has the appropriate permissions, are allowed to proceed. Protecting our API endpoints is a crucial aspect of our security strategy. It not only helps in preserving the integrity and confidentiality of user data but also protects the system from potential attacks such as SQL injections, Cross-Site Scripting (XSS), and others.

```
"currentPassword": "password3",
"newPassword": "UpdatedPassword"
```

es Headers (7) Test Results

Raw Preview Visualize JSON

```
"message": "Invalid token. Please log in again."
```

```
exports.protect = (req, res, next) => {
    let token;


    if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
        token = req.headers.authorization.split(' ')[1];
    }

    if (!token) {
        return res.status(401).send({ message: 'You are not logged in! Please log in to get access.' });
    }

    promisify(jwt.verify)(token, process.env.JWT_SECRET).then(decoded => {

        console.log(decoded.id)

        User.findById(decoded.id, (err, currentUser) => {
            if (err || !currentUser) {
                return res.status(401).send({ message: 'The user belonging to this token no longer exists.' });
            }


            req.user = currentUser;
            next();
        });
    }).catch(err => {
        return res.status(401).send({ message: 'Invalid token. Please log in again.' });
    });
};
```

```
router
.route('/:id')
.get(authController.protect,userController.findUserById)
.delete(authController.protect,authController.restrictToManager(),userController.deleteUserById)
.delete(authController.protect,userController.deleteUserById)
.patch(authController.protect,userController.updateMe);
```

Figure 50 (Backend Code)

Password Reset Processes

In this section, we see the code snippets related to the forgot password and reset password processes that we mentioned earlier. Ensuring a secure password reset process is crucial for several reasons. Firstly, it's vital for protecting user accounts from unauthorized access. If the password reset process is not secure, it can become a vulnerability through which attackers can gain access to user accounts. Securely handling password resets involves verifying the user's identity, typically through email verification, and then allowing them to set a new password in a secure manner. Secondly, a secure password reset process enhances user trust in the system. Knowing that their accounts and personal data are protected, even in the event of forgetting their password, is reassuring for users. In these code sections, we likely implement features such as sending a password reset link or code to the user's registered email address, verifying the reset request, and then securely updating the user's password in the database.

```
exports.forgotPassword = (req, res) => {
  const { email } = req.body;

  User.findUserByEmail(email, async (err, user) => {
      if (err || !user) {
          return res.status(404).send({ message: "User not found with that email." });
      }

      const { resetToken, resetTokenHash, resetTokenExpires } = User.createPasswordResetToken();
      try {
          // Update the user in the database with the reset token and expiry
          await User.updatePasswordResetToken(email, resetTokenHash, resetTokenExpires, (updateErr, updateRes) => {
              if (updateErr) {
                  throw updateErr;
              }
          });
```

```
    // Send the email
    await sendEmail({
        email: user.email,
        subject: 'Your password reset token (valid for 10 min)',
        message
    });

    res.status(200).send({
        status: 'success',
        message: 'Token sent to email!'
    });
} catch (err) {
    // Handle errors
    console.error(err);
    res.status(500).send({ message: 'There was an error processing your request. Please try again later.' });
```

Figure 51 (Backend Code)

Validations

In this section, we find code snippets pertinent to field validations for POST and PATCH requests. Field validation is a critical step before data is committed to the database. It ensures data integrity by verifying that user inputs adhere to the defined format and constraints, thus maintaining the consistency and accuracy of the data.

```
}
setPhone(value) {
    if (value && !validator.isMobilePhone(value, 'tr-TR')) {
        throw new Error("Invalid phone number.");
    }
    this.phone = value;
}


setPasswordHash(value) {
    if (!value) throw new Error("Password hash is required.");
    this.password_hash = value;
}


static isUsernameUnique(username, result) {
  sql.query("SELECT * FROM Users WHERE username = ?", [username], (err, res) => {
      if (err) {
          result(err, null);
          return;
      }
      result(null, { isUnique: res.length === 0 });
  });

if (!validator.isInt(newBookData.number_of_page.toString(), { min: 1 })) {
    return result({ message: "Invalid number of pages" }, null);
}


if (newBookData.description && !validator.isLength(newBookData.description, { max: 2000 })) {
    return result({ message: "Invalid description length" }, null);
}

if (!validator.isLength(newBookData.genre_name, { min: 1, max: 100 })) {
    return result({ message: "Invalid genre name length" }, null);
}

if (!Book.isValidLanguageName(newBookData.language_name)) {

  return result({ message: "Invalid or unsupported language name" }, null);
```

Figure 52 (Backend Code)

Conclusions and Future Work

Our project, encompassing a comprehensive book management and e-commerce platform, represents a significant step forward in digital library systems. We have successfully integrated various modules, including user authentication, payment processing, book and user management, along with advanced features like rating and review systems. These functionalities not only make our system robust but also user-friendly, aligning with our initial objective of enhancing user experience in book exploration and purchase.

Future Enhancements

1. Machine Learning Integration: Implementing machine learning algorithms for personalized book recommendations. This could significantly enhance user engagement by suggesting books based on individual preferences and reading history.

2. Mobile Application Development: While our web platform is robust, developing a mobile application could extend our reach and accessibility. A dedicated app would cater to the

growing number of users who prefer mobile devices for online activities.

3. Internationalization and Localization: Adapting the platform for different languages and regions to cater to a global audience. This would involve not just translating the text but also considering regional differences in book availability and payment options.

4. Advanced Analytics: Incorporating more advanced analytics for both users and administrators. For users, this could involve tracking reading habits and providing insights. For administrators, more in-depth analysis of sales data and user engagement metrics could be valuable.

5. Community Features: Adding more social features such as book clubs, discussion forums, and author events could transform the platform into a more interactive community for book lovers.

6. Sustainability and Scalability: Continuous assessment and improvement in the system's architecture to ensure it remains sustainable and scalable as the user base grows.

References:

1. Microsoft. (2024). Visual Studio. https://visualstudio.microsoft.com/tr/

2. Microsoft. (2024). Visual Studio Code. https://code.visualstudio.com/

3. MySQL. (2024). MySQL Workbench. https://www.mysql.com/products/workbench/

4. Node.js. (2024). Official Website. https://nodejs.org/en

5. Kitapyurdu. (2024). Online Kitap Satış Platformu. https://www.kitapyurdu.com/