Home Assignment Documentation

Luka Petrovic

November 13, 2023

1 Part 1

1.1 Components

1. **Database:**
   - MySQL Server
2. **Data Folder:**
   - Directory for non-database data

1.2 Disaster & Recovery Procedure

1.2.1 Task Scheduler Setup

Open Task Scheduler:
Press Win + R to open the Run dialog.
Type `taskschd.msc` and press Enter.

1.2.2 Create a Basic Task

Click on "Create Basic Task..." in the right-hand Actions pane.

1.2.3 Name and Description

Provide a name and description for the task. Click "Next."

1.2.4 Trigger

Choose "Daily" as the trigger and set the start date and time (e.g., 2:00 AM). Click "Next."

1.2.5 Action

Choose "Start a program" as the action. Click "Next."

1.2.6 Program/script

Browse and locate the `mysqldump.exe` executable. The path might look something like: `C:\Program Files\MySQL\MySQL Server X.X\bin\mysqldump.exe`.
If the MySQL bin directory is not in the system's PATH, provide the full path to `mysqldump.exe`.

1.2.7 Add arguments

In the "Add arguments" field, provide the necessary arguments for `mysqldump`:

```
-u luka -p paracin95 motuNovu > C:\path\to\backup\daily_backup.sql
```

1.3 Easily Recover Data for Last Month

1.3.1 Daily Backup Script

Assuming a daily backup script using Task Scheduler was set, we ensure the script looks like this:

```
powershell -Command "mysqldump -u luka -p paracin95 motuNovu > C:\path\to\backup\daily_backup_$(Get-Date
    -Format 'yyyy-MM-dd').sql"
```

1.3.2 Recovery Script for Last Month

To recover data for the last month, use the following PowerShell command:

```
powershell -Command "mysql -u luka -p paracin95 motuNovu < C:\path\to\backup\daily_backup_$(Get-Date (Get
    -Date).AddMonths(-1) -Format 'yyyy-MM-dd').sql"
```

1.4 Rebuild Data at the Exact Date for Last Year

1.4.1 Full Yearly Snapshot Script

Adjust the yearly backup script to include the date in the backup file name:

```
powershell -Command "mysqldump -u luka -p paracin95 motuNovu > C:\path\to\backup\yearly_backup_$(Get-Date
     -Format 'yyyy-MM-dd').sql"
```

1.4.2 Rebuild Script for Last Year

To rebuild data for the exact date last year, the following PowerShell command was used:

```
powershell -Command "mysql -u luka -p paracin95 motuNovu < C:\path\to\backup\yearly_backup_$(Get-Date (
    Get-Date).AddYears(-1) -Format 'yyyy-MM-dd').sql"
```

1.5 Recover Data Forever with a Reasonable Time Interval

Ensure the backup strategy is robust, and we are retaining backups for an extended period. Adjust the time intervals in your backup scripts accordingly. For example, we might want to keep monthly or yearly backups indefinitely. Regularly we test the recovery process to ensure the backups are viable and the recovery scripts are functioning correctly.

2 Part 2

2.1 Create Database and Tables

Firstly, MySQL database and tables were created, then the following SQL queries were executed:

```sql
CREATE DATABASE motunovu;

USE motunovu;

-- Create Users table
CREATE TABLE users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  username VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL
);

-- Create Customers table
CREATE TABLE customers (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL
);

-- Create Products table
CREATE TABLE products (
  id INT PRIMARY KEY AUTO_INCREMENT,
```

```
  name VARCHAR(255) NOT NULL,
  price DECIMAL(10, 2) NOT NULL
);
```

2.1.1 Database Connection

Connection to the MySQL database was established using the 'mysql' module.

```
const pool = mysql.createPool({
  host: 'localhost',
  user: 'luka',
  database: 'motunovu',
  password: 'paracin95',
  port: 3306,
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});
```

2.2 APIs and CRUD Operations

2.2.1 Retrieve All Products

The following code snippet show the API endpoint that retrieves all products from the database(beside this one, all four CRUD operations are implemented).

```
// Retrieve all products
router.get('/products', async (req, res) => {
  try {
    const [rows] = await pool.query('SELECT * FROM products');
    res.json(rows);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

2.3 Frontend Integration

2.3.1 Fetch Products in Frontend

This code snippet shows how to fetch products from the backend in the frontend using Axios(library for making HTTP requests from a web browser or a Node. js server).

```
const fetchProducts = async () => {
  try {
    const response = await axios.get('http://localhost:3001/api/products');
    console.log('Products:', response.data); // Log products to the console
    setProducts(response.data);
  } catch (error) {
    console.error('Error fetching products:', error);
  }
};
```

Another example is the function *fetchCustomerDetails*, which fetches all the customers through the API:

```
  const fetchCustomerDetails = async () => {
    try {
      const response = await axios.get(`http://localhost:3001/api/customers`);
    } catch (error) {
      console.error('Error fetching customer details:', error);
    }
  };
```

And then shows them in the list, which is shown in the following section.

2.3.2 HTML Part - Display Customer List

This HTML code displays a list of all customers (which were initally added in the database).

```
<div className="customer-list">
  <h2>Customer List</h2>
  <button onClick={fetchCustomerDetails}>List Customers</button>
  <ul>
    {customers.map((customer) => (
      <li key={customer.id}>
        {`Name: ${customer.name}`}
      </li>
    ))}
  </ul>
</div>
```

3 Part 3

3.1 Database Setup

3.1.1 Create Database and Tables

In this part we again create the necessary tables for Products, Customers, and Orders with one-to-many relationships (in this one, we have to take care of the One-To-Many relationships between them and use foreign keys).

```
-- Create Orders table with foreign keys
CREATE TABLE orders (
  id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  FOREIGN KEY (customer_id) REFERENCES customers(id)
);

-- Create Order_Products table with foreign keys
CREATE TABLE order_products (
  order_id INT,
  product_id INT,
  quantity INT,
  PRIMARY KEY (order_id, product_id),
  FOREIGN KEY (order_id) REFERENCES orders(id),
  FOREIGN KEY (product_id) REFERENCES products(id)
);
```

3.1.2 User Roles

Then, we create a 'users' table to manage user roles. (for now, two roles were created *Admin*/Manager and *User*)

```
-- Create Users table with roles
CREATE TABLE users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  username VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  role ENUM('User', 'Admin') NOT NULL
);
```

3.2 User Authentication

Implement user authentication for different roles. In this implementation, session-based authentication was employed instead of JWT tokens, providing a server-side mechanism where user sessions are maintained on the server, and authentication is managed through session cookies, as opposed to JWT tokens that store user information client-side and require stateless verification through cryptographic signatures. (both can be done in the same manner, but this one was chosen for this task)

3.2.1 Backend Login Endpoint

Creation of a new route for user authentication:

```
app.post('/api/login', async (req, res) => {
  const { username, password } = req.body;

  try {
    const connection = await pool.getConnection();
    const [rows] = await connection.query(
      'SELECT * FROM users WHERE username = ? AND password = ?',
      [username, password]
    );

    if (rows.length > 0) {
      res.json({ success: true, user: rows[0] });
    } else {
      res.json({ success: false, message: 'Invalid credentials' });
    }

    connection.release();
  } catch (error) {
    console.error('Error during login:', error);
    res.status(500).json({ success: false, message: 'Internal server error' });
  }
});
```

3.2.2 Frontend Integration

Integration of the authentication into the frontend:

```
// Frontend login check
const handleLogin = (username, role) => {
  setUser(username);
  setRole(role);
  setAuthenticated(true);
};
```

In this example, the *handleLogin* function is used in the Login component:

```

const handleLogin = async () => {
        try {
            const response = await axios.post('http://localhost:3001/api/login', {
                username,
                password,
            });

            if (response.data && response.data.success) {
                onLogin(username);
            } else {
                alert('Invalid credentials');
            }
        } catch (error) {
            console.error('Error during login:', error);
            alert('Error during login');
        }
    };

    return (
        <div className="login">
            <h2>Login</h2>
            <label htmlFor="username">Username:</label>
            <input
                type="text"
                id="username"
                value={username}
                onChange={(e) => setUsername(e.target.value)}
            />
            <label htmlFor="password">Password:</label>
            <input
                type="password"
                id="password"
```

```
                value={password}
                onChange={(e) => setPassword(e.target.value)}
            />
            <button onClick={handleLogin}>Login</button>
        </div>
    );
};
```

Once the user is authenticated, it checks the user role (admin or user) and conditionally renders the appropriate dashboard (AdminDashboard or UserDashboard). The handleLogout function is used to log out the user:

```
    div className="app">
      {!authenticated ? (
        <Login onLogin={handleLogin} />
      ) : (
        <div>
          {user === 'admin' ? (
            <>
              <h1>Admin Management Page</h1>
      ) : (
            <>
              {/* New section for non-admin users */}
              <h1>User Management Page</h1>
      )}
      <button onClick={handleLogout}>Logout</button>
    </div>
</div>
```

3.3 CRUD Operations

CRUD operations for Products, Customers, and Orders based on user roles were implemented as well. Some of the example code snippets are shown:

3.3.1 Delete a Product by ID

Allow Managers(Admin in our case) to delete a product.

```
router.delete('/products/:id', async (req, res) => {
  const productId = req.params.id;

  try {
    const [result] = await pool.execute('DELETE FROM products WHERE id = ?', [productId]);

    if (result && result.affectedRows > 0) {
      res.json({ message: 'Product deleted successfully' });
    } else {
      res.status(404).json({ error: 'Product not found' });
    }
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

3.3.2 Update a Product by ID

Allow Managers/Admin to update a product.

```
router.put('/products/:id', async (req, res) => {
  const productId = req.params.id;
  const { name, price } = req.body;

  try {
    const [result] = await pool.execute('UPDATE products SET name = ?, price = ? WHERE id = ?', [name,
     price, productId]);

    if (result && result.affectedRows > 0) {
      res.json({ message: 'Product updated successfully' });
```

```
    } else {
      res.status(404).json({ error: 'Product not found' });
    }
  } catch (error) {
    console.error('Error updating product:', error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

3.3.3 Add a New Order

Allow *User* to create new orders.

```
router.post('/orders', async (req, res) => {
  const { customerId, productId } = req.body;

  try {
    const [orderResult] = await pool.execute('INSERT INTO Orders (customer_id) VALUES (?)', [customerId])
     ;
    const orderId = orderResult.insertId;

    await pool.execute('INSERT INTO order_products (order_id, product_id, quantity) VALUES (?, ?, ?)', [
     orderId, productId, 1]);

    res.json({ message: 'Order created successfully' });
  } catch (error) {
    console.error('Error creating order:', error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

3.3.4 Delete a Customer by ID

Allow Managers to delete a customer.

```
router.delete('/customers/:id', async (req, res) => {
  const customerId = req.params.id;

  try {
    const [orders] = await pool.execute('SELECT id FROM orders WHERE customer_id = ?', [customerId]);

    if (orders.length > 0) {
      await pool.execute('DELETE FROM orders WHERE customer_id = ?', [customerId]);
    }

    const [result] = await pool.execute('DELETE FROM customers WHERE id = ?', [customerId]);

    if (result && result.affectedRows > 0) {
      res.json({ message: 'Customer deleted successfully' });
    } else {
      res.status(404).json({ error: 'Customer not found' });
    }
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

4 Conclusion

In conclusion, this document provides a concise overview of the key components and operations involved in the development of the application. The examples and snippets covered include the establishment of a relational database with tables for users, customers, products, and orders, accompanied by CRUD operations. The integration of user roles and authentication mechanisms is highlighted, demonstrating the implementation of two roles: User and Manager. While the presented code snippets serve as a foundational guide, it is important to note that the complete and detailed codebase is accessible on GitHub

for further exploration. Additionally, this project offers ample opportunities for enhancement, such as addressing various use case scenarios, handling potential errors, and bolstering security measures, including the adoption of more sophisticated techniques like JWT tokens. Continuous refinement and adaptation to evolving requirements can contribute to the robustness and scalability of the application.