Architectural Pattern: Model–View–Controller (MVC) for BUP Alumni System

1. Introduction

The BUP Alumni System is a web-based platform designed to connect the former students of Bangladesh University of Professionals (BUP). It allows users to register, log in, update profiles, participate in events, and communicate within the alumni network.
To ensure that the system remains organized, maintainable, and scalable, a well-defined architectural pattern is required.

The project uses the following technology stack:
● **Languages:** HTML, CSS, JavaScript
● **Frontend Framework:** React.js with Tailwind CSS
● **Backend Framework:** Node.js with Express.js
● **Database:** MySQL
● **Editor:** Visual Studio Code

Considering the functional requirements and chosen technologies, the **Model–View–Controller (MVC)** architectural pattern is the most suitable for this project.

2. What is an Architectural Pattern?

An architectural pattern provides a structured solution to recurring software design problems. It defines how system components are organized and how they interact to fulfill functional and non-functional requirements.
By introducing clear separation of concerns, architectural patterns help improve scalability, maintainability, and team collaboration.

Common architectural patterns include:
● Model–View–Controller (MVC)
● Three-Tier Architecture
● Client–Server Architecture
● Event-Driven Architecture

3. Why an Architectural Pattern is Needed for Our Project

For the **BUP Alumni System**, an architectural pattern provides a solid foundation and structure. Key reasons include:

1. **Separation of Concerns:** Clearly divides responsibilities among Model, View, and Controller.
2. **Maintainability:** Enables developers to modify one component without breaking others.
3. **Scalability:** Supports easy addition of new modules or features.
4. **Reusability:** The same Models and Controllers can serve multiple Views (web, mobile, admin).
5. **Security:** Prevents direct client access to sensitive data or business logic.

The MVC pattern provides a clean organization of code that fits the BUP Alumni System's modular and evolving nature.

4. What is Model–View–Controller (MVC)?

The MVC pattern divides an application into three main interconnected components, each handling a specific aspect of the application's functionality.

Model

- Represents the data and the business logic of the system.
- Interacts directly with the database and manages operations like data retrieval, insertion, and validation.
- Technologies: MySQL (Database)
- Example: Handles user login verification, registration data, and alumni profiles.

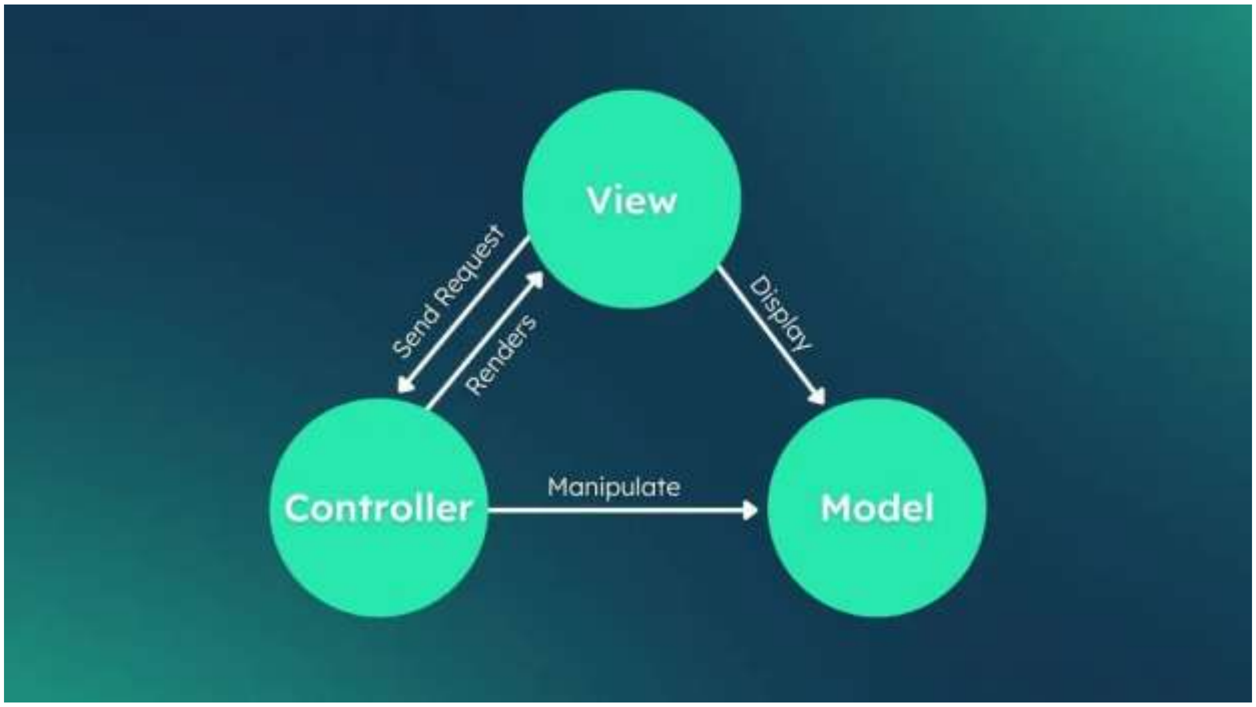
View

- Responsible for the user interface (UI) and presentation of information.
- Displays data fetched from the Controller and collects user input.
- Technologies: React.js, Tailwind CSS, HTML, CSS, JavaScript.
- Example: Login form, profile dashboard, event list page.

Controller

- Acts as an intermediary between the View and the Model.
- Receives input from the user (via View), processes it using business logic, and updates the Model or View accordingly.
- Technologies: Node.js, Express.js.
- Example**:** The loginController verifies credentials, fetches data from the Model, and returns a response to the View.

5. MVC Architecture Diagram



6. How MVC Works

The MVC pattern ensures smooth interaction between the user interface, business logic, and database. Here’s how it works in the BUP Alumni System:

Step 1 — User Interaction (View):
The user interacts with the React.js frontend, such as entering login details and submitting the form.

Step 2 — Controller Request Handling:
The Controller (Node.js/Express.js) receives the request and determines the intended operation (e.g., authentication).

Step 3 — Model Processing:
The Controller communicates with the Model to execute business logic — for instance, verifying credentials by querying the MySQL database.

Step 4 — Database Operation:
The Model retrieves or updates data in the database and returns the result to the Controller.

Step 5 — Response to View:
The Controller sends the processed data or response (e.g., “Login Successful”) back to the View for display.

This workflow maintains a clean separation of concerns and ensures each component operates independently but cohesively.

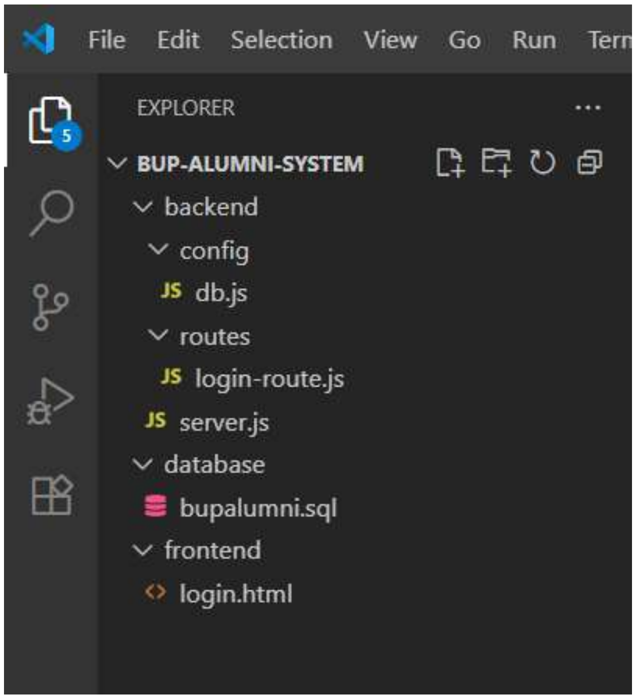
7. Example File Structure (MVC Implementation)

```
bup-alumni-system/
│
├── backend/
│   ├── server.js
│   │
│   ├── config/
│   │   └── db.js
│   │
│   ├── models/
│   │   ├── userModel.js
│   │   ├── eventModel.js
│   │   └── alumniModel.js
│   │
│   ├── controllers/
│   │   ├── loginController.js
│   │   ├── signupController.js
│   │   ├── profileController.js
│   │   └── eventController.js
│   │
│   ├── routes/
│   │   ├── login-route.js
│   │   ├── signup-route.js
│   │   ├── profile-route.js
│   │   └── event-route.js
│   │
│   └── frontend/
│      ├── login.html
│      ├── signup.html
```

```
|       ├── profile.html
|       └── event.html
|
├── database/
|   └── bupalumni.sql
|
└── css/
|    ├── login.css
|    ├── signup.css
|    ├── user_profile.css
└── package.json
```

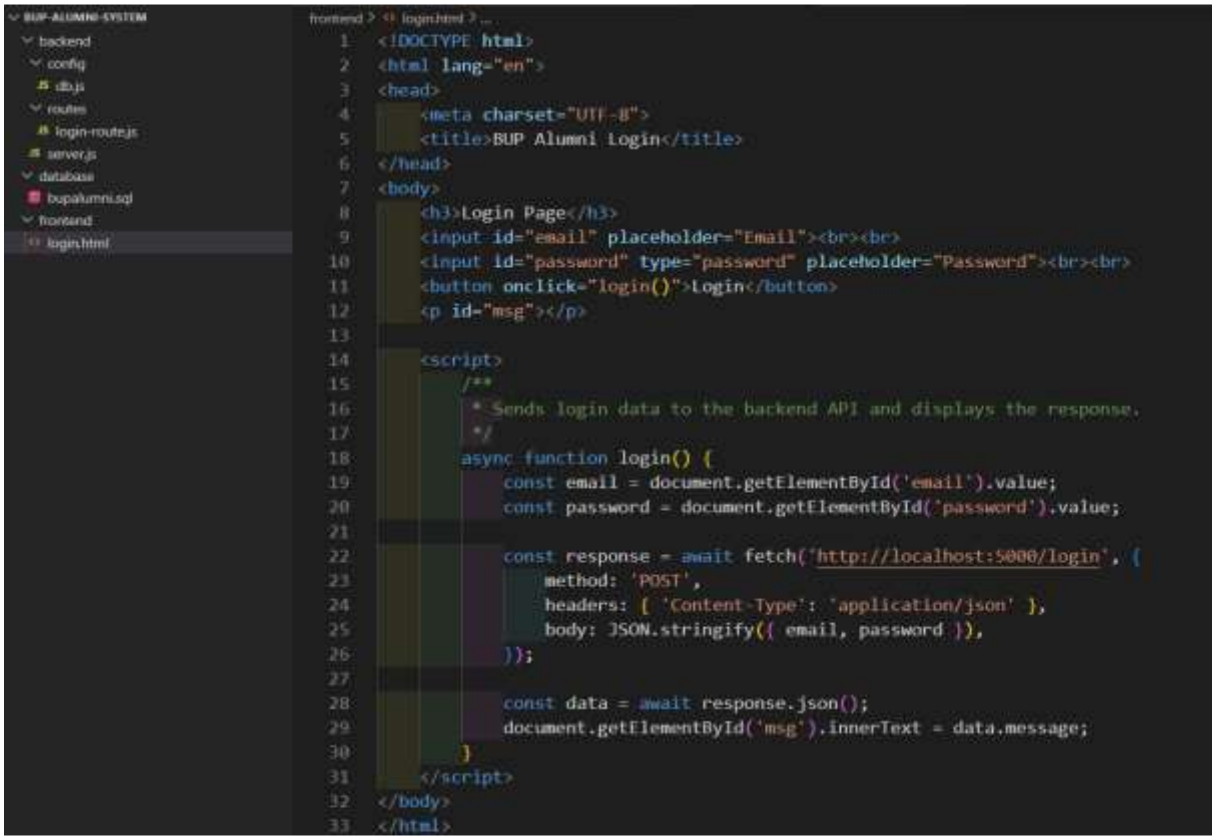
8. Step-by-Step Implementation Example: (Login Implementation)

Step 1: Create Project Folder
Action: Create a main folder named bup-alumni-system in VS Code.
Description: This folder will hold all files — frontend, backend, and database setup.
Folder Structure:



```
BUP-ALUMNI-SYSTEM
  backend
    config
      db.js
    routes
      login-route.js
    server.js
  database
    bupalumni.sql
  frontend
    login.html
```

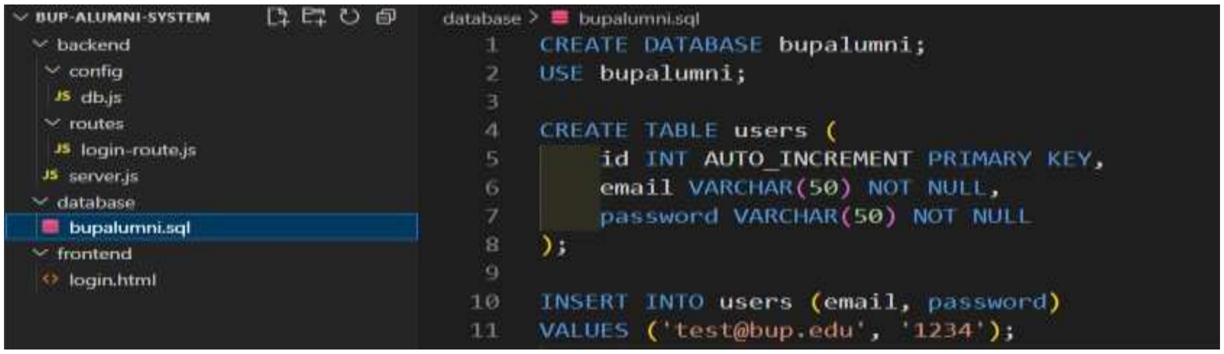
Step 2: Create the Frontend (User Interface)
File: frontend/login.html
Description: This page collects user input (email and password) and sends it to the backend.



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>BUP Alumni Login</title>
</head>
<body>
    <h3>Login Page</h3>
    <input id="email" placeholder="Email"><br><br>
    <input id="password" type="password" placeholder="Password"><br><br>
    <button onclick="login()">Login</button>
    <p id="msg"></p>

    <script>
        /**
         * Sends login data to the backend API and displays the response.
         */
        async function login() {
            const email = document.getElementById('email').value;
            const password = document.getElementById('password').value;

            const response = await fetch('http://localhost:5000/login', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ email, password }),
            });

            const data = await response.json();
            document.getElementById('msg').innerText = data.message;
        }
    </script>
</body>
</html>
```

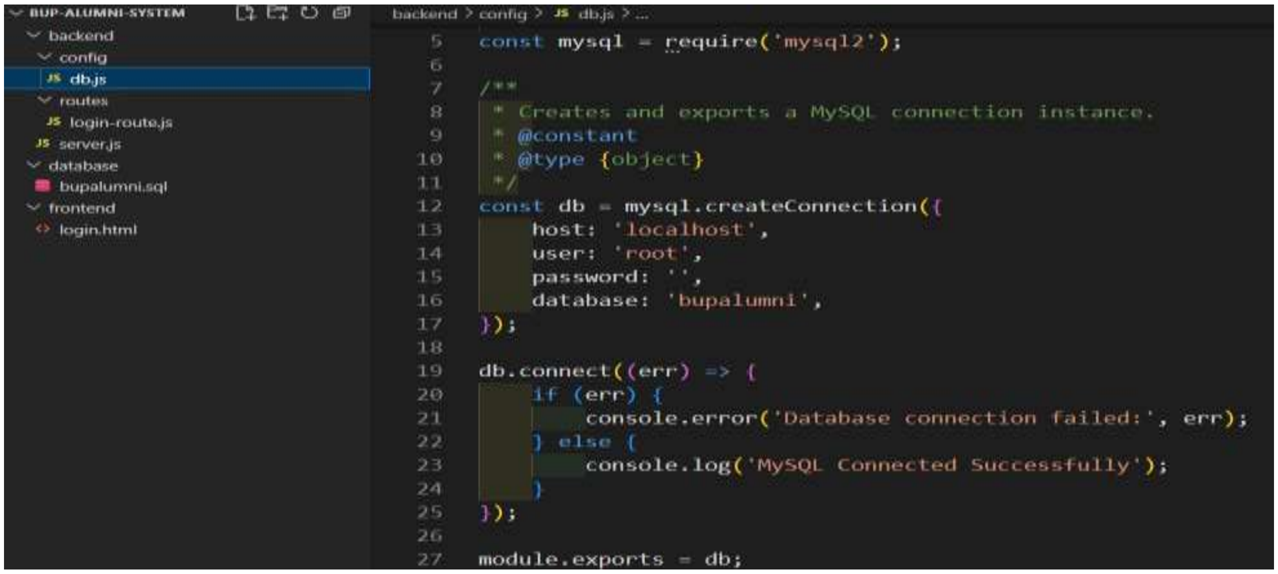
Step 3: Set Up the Database
File: database/bupalumni.sql
Description: This SQL file creates the database and the users table for login.
Steps:

- Open XAMPP or MySQL Workbench.
- Create a new database named bupalumni.
- Run this SQL code.



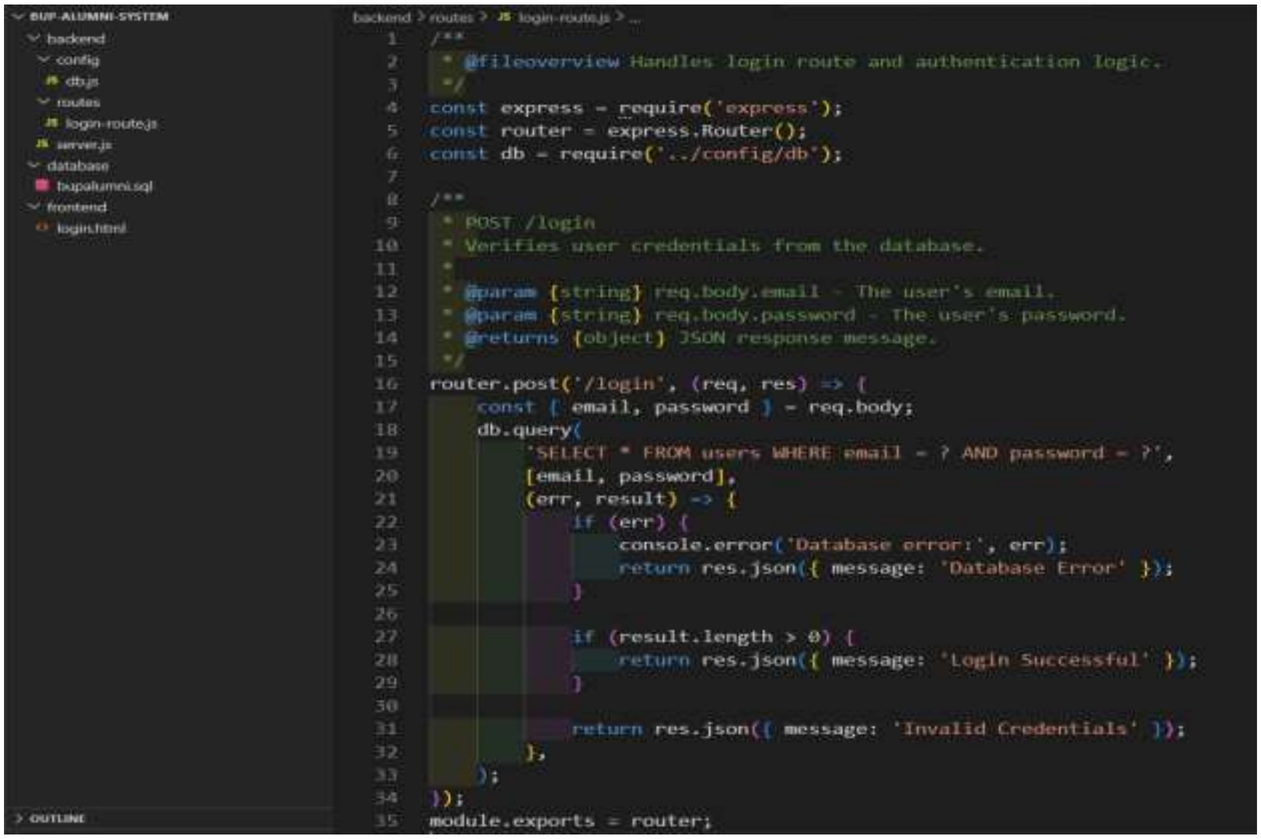
```sql
CREATE DATABASE bupalumni;
USE bupalumni;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL
);

INSERT INTO users (email, password)
VALUES ('test@bup.edu', '1234');
```

Step 4: Set Up Backend Configuration
File: backend/config/db.js
Description: This file connects the Node.js backend to the MySQL database.



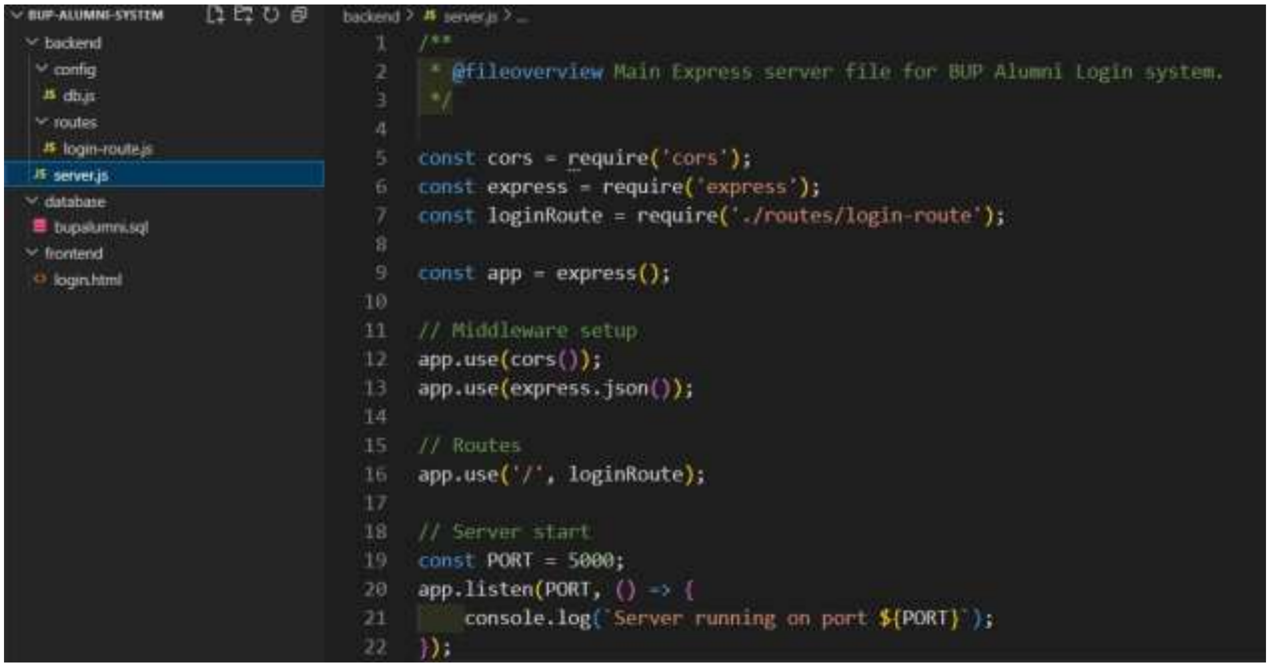
```javascript
const mysql = require('mysql2');

/**
 * Creates and exports a MySQL connection instance.
 * @constant
 * @type {object}
 */
const db = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'bupalumni',
});

db.connect((err) => {
    if (err) {
        console.error('Database connection failed:', err);
    } else {
        console.log('MySQL Connected Successfully');
    }
});

module.exports = db;
```

Step 5: Create the Login Route
File: backend/routes/login-route.js
Description: This file handles the login request sent from the frontend.



```javascript
/**
 * @fileoverview Handles login route and authentication logic.
 */
const express = require('express');
const router = express.Router();
const db = require('../config/db');

/**
 * POST /login
 * Verifies user credentials from the database.
 *
 * @param {string} req.body.email - The user's email.
 * @param {string} req.body.password - The user's password.
 * @returns {object} JSON response message.
 */
router.post('/login', (req, res) => {
    const { email, password } = req.body;
    db.query(
        'SELECT * FROM users WHERE email = ? AND password = ?',
        [email, password],
        (err, result) => {
            if (err) {
                console.error('Database error:', err);
                return res.json({ message: 'Database Error' });
            }

            if (result.length > 0) {
                return res.json({ message: 'Login Successful' });
            }

            return res.json({ message: 'Invalid Credentials' });
        },
    );
});
module.exports = router;
```

Step 6: Create the Main Server File
File: backend/server.js
Description: This is the main file that runs the backend server.



```
/**
 * @fileoverview Main Express server file for BUP Alumni Login system.
 */

const cors = require('cors');
const express = require('express');
const loginRoute = require('./routes/login-route');

const app = express();

// Middleware setup
app.use(cors());
app.use(express.json());

// Routes
app.use('/', loginRoute);

// Server start
const PORT = 5000;
app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

Step 7: Install Required Packages
Description: These packages help connect and run the backend server.
Steps:

- Open VS Code Terminal.
- Run the following
  commands: cd backend
  npm init -y
  npm install express mysql2 cors

Step 8: Run the Backend Server
Description: This starts your backend so that the frontend can send login requests.
Steps: In VS Code Terminal, run:
node server.js

9. Cohesion and Coupling in the BUP Alumni System

Cohesion:
Cohesion refers to how closely the parts within a single component or module work together to achieve a single purpose. In a well-designed MVC (Model-View-Controller) architecture, each layer has a specific and clearly defined responsibility, leading to high cohesion.

In Our Project (MVC-based BUP Alumni System):
The system is organized into three main layers — Model, View, and Controller, each having a distinct role:

- Model Layer: Manages all data-related logic, such as database operations and data structures.
  Example: The UserModel.js or db.js file handles alumni information (name, email, password, profession) and database queries, without managing UI or user interactions.
- View Layer: Responsible for the presentation and user interface.
  Example: The login.html or profile.html files display data to users and take their input, but do not contain any business or database logic.
- Controller Layer: Acts as a bridge between the View and Model layers, handling user requests, processing data, and deciding what to show in the View.
  Example: The loginController.js file receives login input from the view, validates it, and interacts with the Model to verify user credentials — it doesn't handle how the data is stored or displayed.

This clear separation ensures that every layer has one focused purpose and its internal elements work together toward that goal, achieving high cohesion

Coupling:
Coupling refers to the degree of dependency between different modules or layers. In the MVC pattern, low (loose) coupling is achieved because the Model, View, and Controller communicate through well-defined interfaces rather than being tightly connected.

In Our Project (MVC-based BUP Alumni System):

- The View layer interacts only with the Controller, not directly with the Model or database.
  Example: The login.html page sends user credentials to the controller through a form or API call, rather than accessing data directly.
- The Controller interacts with the Model to fetch or update data and then sends the processed results back to the View.
  Example: The loginController.js calls methods from UserModel.js to verify login details, without knowing database implementation details.

- The Model layer is independent of how data is presented or how users interact with the system.

If any layer changes — for example, switching from MySQL to MongoDB — only the Model code (database logic) needs to be updated. The Controller and View layers remain unaffected, demonstrating loose coupling.

10. Advantages of MVC Architecture

● **Separation of Concerns:** Each component has a clear and distinct purpose.
● **Reusability:** Models and Controllers can serve multiple user interfaces.
● **Maintainability:** Updates or fixes can be made in isolated parts without affecting the rest.
● **Scalability:** Easier to extend with new features or platforms.
● **Parallel Development:** Different teams can work simultaneously on Models, Views, and Controllers.

11. Disadvantages of MVC Architecture

● **Increased Initial Setup:** Requires structured configuration of multiple folders and files.
● **Learning Curve:** Beginners may find understanding MVC flow challenging at first.
● **Overhead for Small Projects:** For very simple applications, MVC can add unnecessary complexity.

12. Conclusion

The Model–View–Controller (MVC) architecture provides a robust, modular, and scalable foundation for the BUP Alumni System.
By separating the application into Models, Views, and Controllers, the project achieves clarity, maintainability, and long-term adaptability.
With React.js as the View, Node.js/Express.js as the Controller, and MySQL as the Model, this architecture perfectly aligns with the project's design goals and ensures a clean, extensible system structure for future growth.

13. References

● IBM — *What is Model–View–Controller (MVC)?*
https://www.ibm.com/think/topics/model-view-controller
● GeeksforGeeks — *Model View Controller (MVC) Architecture*
https://www.geeksforgeeks.org/mvc-design-pattern/