**EX NO: 9**

**DATE:**

**IMPROVE THE REUSABILITY AND MAINTAINABILITY OF THE SOFTWARE SYSTEM BY APPLYING APPROPRIATE DESIGN PATTERNS.**

**AIM:**

To improve the reusability and maintainability of the software system by applying appropriate design patterns.

**DESIGN PATTERNS USED:**

1. **Model-View-Controller (MVC) Pattern**

**Overview:**

The Model-View-Controller (MVC) pattern is applied in the project to separate concerns, making the code more modular, scalable, and easier to maintain.

**Implementation:**

* **Model (Database Operations - MongoDB):**
  + Handles data logic and database interactions.
  + Defined using MongoDB collections (students, faculty, courses, marks).

**Example:**const { MongoClient } = require('mongodb');

class Database {

constructor(uri) {

this.client = new MongoClient(uri);

}

async connect() {

await this.client.connect(); }

async getCollection(name) {

return this.client.db("sis").collection(name); }}

* **View (API Responses - Express.js):**
  + Defines how data is presented to the user.
  + Returns JSON responses in Express API endpoints.

**Example:** app.get('/students', async (req, res) => {

const students = await database.getCollection("students").find().toArray();

res.json(students);});

* **Controller (Business Logic - API Handlers):**
  + Handles HTTP requests and interacts with models.
  + Implements validation and processing logic before data is stored or retrieved.

**Example:** class StudentController {

constructor(db) {

this.db = db;

}

async getStudents(req, res) {

try {

const students = await this.db.getCollection("students").find().toArray();

res.json(students);

} catch (error) {

res.status(500).json({ message: "Error fetching students", error }); } }}

**Benefits of Using MVC:**

* **Separation of Concerns:** Makes code modular.
* **Reusability:** Components can be reused across different parts of the application.
* **Maintainability:** Easier debugging and updates.

1. **Adapter Pattern**

**Overview:**

The Adapter pattern is used to make the database interactions modular and interchangeable, allowing easy changes in database configuration.

**Implementation:**

* **Problem:** Directly using MongoClient in multiple API routes resulted in repeated code and tight coupling.
* **Solution:** Created a **Database Adapter** class to abstract database operations.

**Database Adapter Implementation:**

class DatabaseAdapter {

constructor(uri) {

this.client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });

}

async connect() {

if (!this.client.topology || !this.client.topology.isConnected()) {

await this.client.connect();

}

return this.client.db("sis");

}

async getCollection(collectionName) {

const db = await this.connect();

return db.collection(collectionName);

}}

**Usage in Controllers:**

const database = new DatabaseAdapter("your\_mongo\_uri");

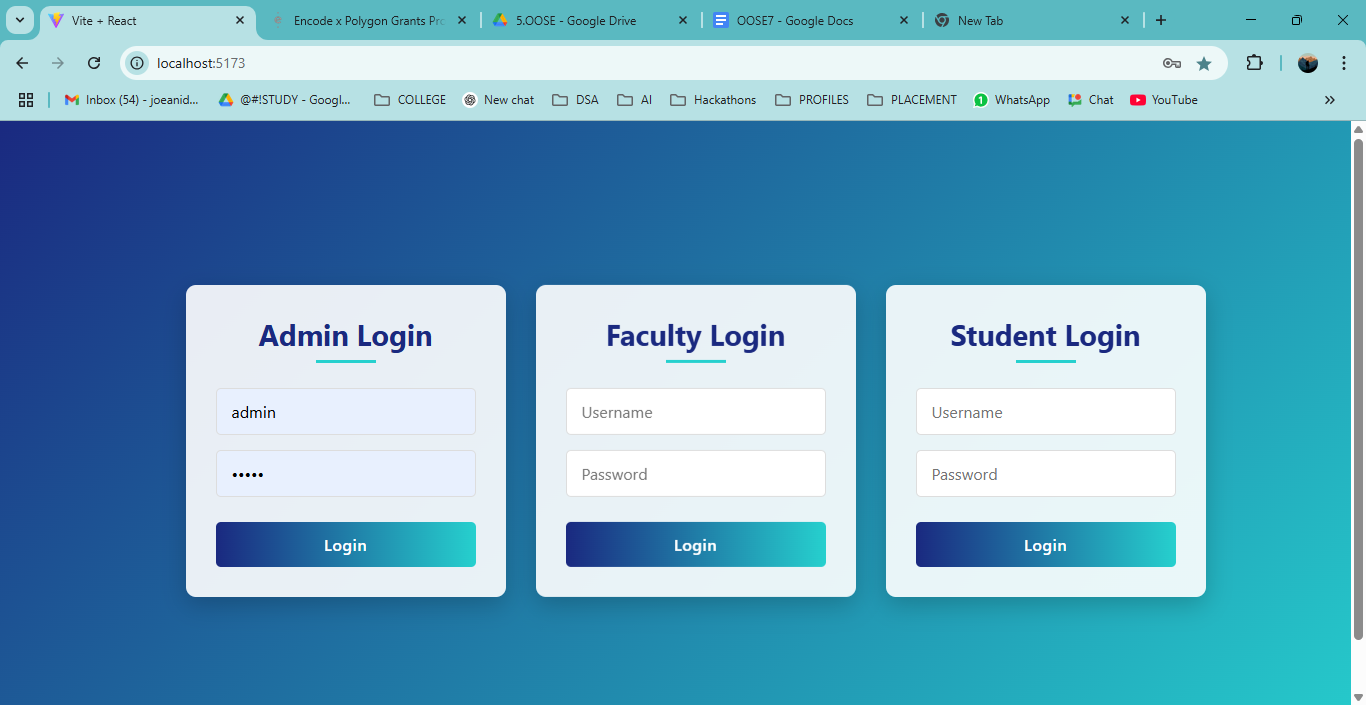
const studentController = new StudentController(database);

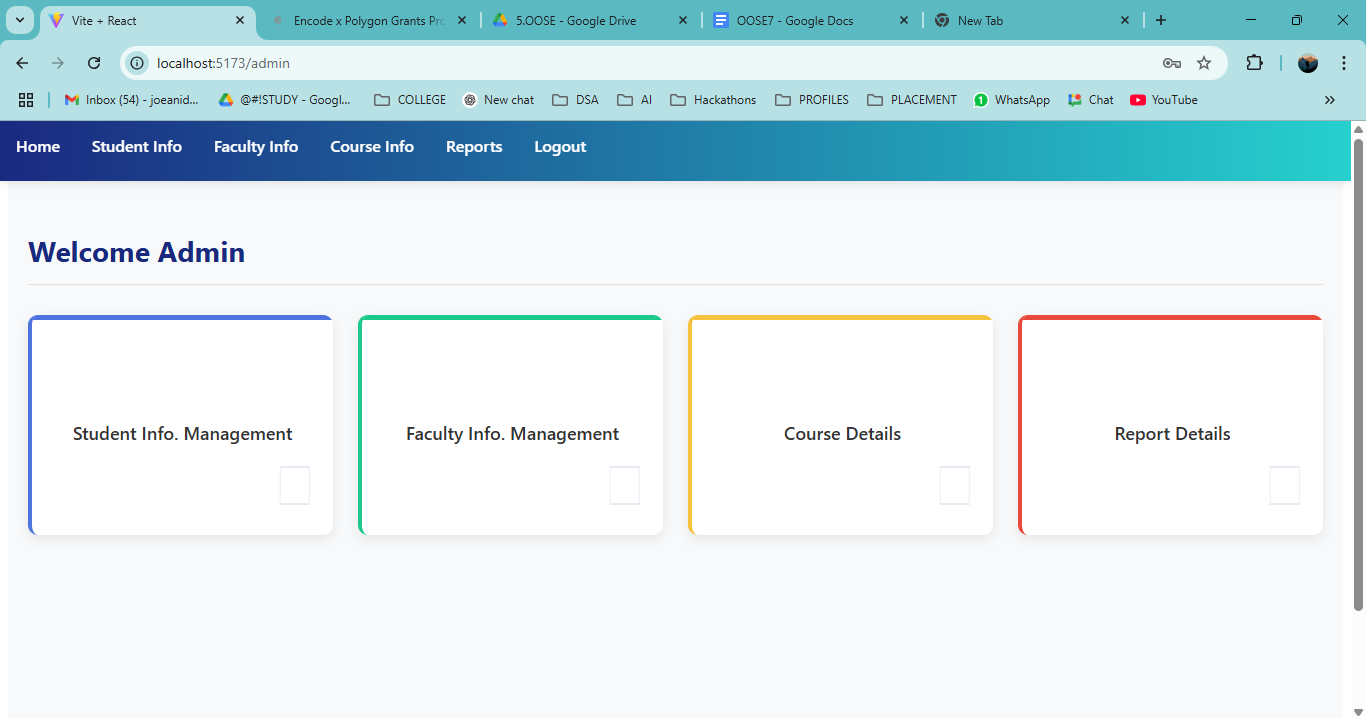
app.get('/students', studentController.getStudents.bind(studentController));

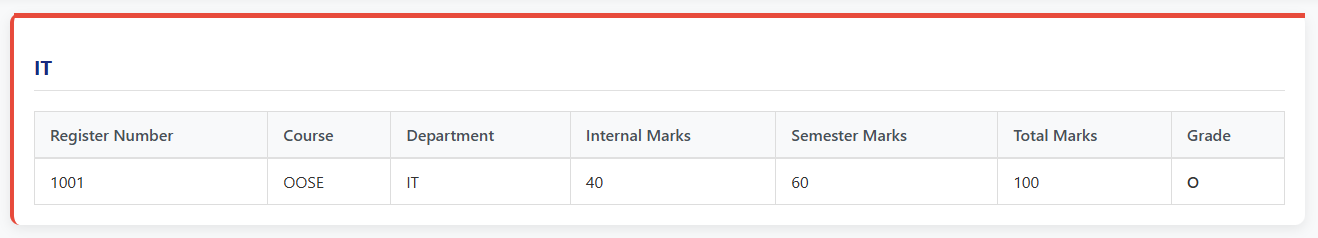
**Benefits of Using Adapter Pattern:**

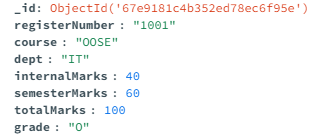
* **Decoupling:** Database logic is separated from API routes.
* **Flexibility:** Can switch databases easily (e.g., MongoDB to PostgreSQL).
* **Reusability:** Database logic can be reused across multiple controllers.

**OUTPUT:**

****

****





**RESULT:**