**MongoDB Module End Assignment**

**--------------------------------------------------------------------------------------------------**

**Task 1: Understanding Databases (5 Marks)**

* **Explain the key differences between SQL & NoSQL databases.**

Databases are fundamental components of modern applications that store and manage data efficiently. Two major types of databases are SQL (Structured Query Language) databases and NoSQL (Not Only SQL) databases. Each has unique characteristics, strengths, and use cases.

**Key Differences Between SQL & NoSQL**

| **Feature** | **SQL Databases** | **NoSQL Databases** |
| --- | --- | --- |
| Data Model | Relational (tables with rows and columns) | Non-relational (documents, key-value, graph) |
| Schema | Rigid schema; predefined structure | Flexible schema; dynamic structure |
| Scalability | Vertical scaling | Horizontal scaling |
| Query Language | Structured Query Language (SQL) | Varies (MongoDB uses queries similar to JSON) |
| Transactions | ACID-compliant (strong consistency) | Often eventual consistency; BASE model |
| Examples | MySQL, PostgreSQL, Oracle | MongoDB, Cassandra, Redis, CouchDB |

SQL databases are ideal for applications where data integrity, complex queries, and strict schema constraints are essential. On the other hand, NoSQL databases are preferred when flexibility, high scalability, and performance for unstructured data are priorities.

* **Define Schema, Document, and Collection in MongoDB.**

In MongoDB (a NoSQL database), three key concepts are:

**Schema**: In MongoDB, the schema is optional and defines the structure of documents in a collection. Unlike SQL, schema enforcement is flexible, allowing documents in the same collection to have different fields.

**Document**: The basic unit of data in MongoDB, stored in BSON format (binary JSON). Each document is analogous to a row in SQL and can contain nested data.

**Collection**: A grouping of MongoDB documents, equivalent to a table in SQL. Collections do not enforce a schema by default.

**Provide examples of scenarios where SQL is preferred and where NoSQL is better.**

**A short 400-500 word report on SQL vs NoSQL with real-world use cases.**

**Real-World Use Cases**

**When SQL is Preferred**:

* **Banking and Financial Applications**: Data accuracy and integrity are critical. SQL's ACID compliance ensures transactional reliability.
* **Inventory Management**: Structured schema, strong relationships, and consistent records are crucial.
* **Customer Relationship Management (CRM)**: Well-defined relationships between customers, products, and sales data are best handled in a relational format.

**When NoSQL is Preferred**:

* **Real-Time Analytics**: Applications like social media feeds or clickstream data benefit from the scalability and speed of NoSQL.
* **Content Management Systems (CMS)**: Flexible and diverse content structures make MongoDB ideal.
* **E-Commerce Platforms**: Products with varying attributes, dynamic schema, and fast search requirements are well-suited to NoSQL.

In conclusion, SQL and NoSQL databases serve different purposes. SQL offers robust structure and data integrity, ideal for transactional systems. NoSQL provides flexibility, scalability, and performance, making it suitable for dynamic and large-scale applications. Choosing between the two depends on the specific needs of the project.

**Task 2: Setting Up MongoDB (5 Marks)**

* **Install MongoDB Compass and connect to a local MongoDB database.**
* **Create a free MongoDB Atlas cloud database and connect it using Mongoose.**
* **Take a screenshot showing a successful connection in Compass or Atlas.**

**db.js code**const mongoose = require("mongoose");

const mongoURI = "mongodb+srv://tallalokeshedu:OVw4arCeVirPEiOQ@cluster0.zta3d3c.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0";

mongoose.connect(mongoURI, {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() => console.log("MongoDB Connected Successfully"))

.catch(err => console.error("MongoDB Connection Error:", err));

module.exports = mongoose;

**index.js**

const express = require("express");

require("./db");

const app = express();

app.use(express.json());

app.get("/", (req, res) => {

res.send("Hello from MongoDB Connected App!");

});

const PORT = process.env.PORT || 5000;

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

**Screenshots:**

A black background with white text

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**Task 3: Creating Schema & Model in Mongoose (5 Marks)**

* **Define a Schema for a User collection with fields:**
  + **name (String, required)**
  + **email (String, required, unique)**
  + **age (Number, optional)**
* **Use Mongoose Model to interact with the collection.**

**Models/User.js**

const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({

name: {

type: String,

required: true,

},

email: {

type: String,

required: true,

unique: true,

},

age: {

type: Number,

}

});

const User = mongoose.model("User", userSchema);

module.exports = User;

**Task 4: CRUD Operations Using Mongoose (10 Marks)**

* **Implement Create, Read, Update, Delete (CRUD) operations using Express & Mongoose.**
* **Use Mongoose Queries in Routes for handling requests.**

**Routes/userRoutes.js**

const express = require("express");

const User = require("../models/User");

const router = express.Router();

router.post("/users", async (req, res) => {

try {

const newUser = new User(req.body);

await newUser.save();

res.status(201).json(newUser);

} catch (error) {

res.status(400).json({ error: error.message });

}

});

router.get("/users", async (req, res) => {

try {

const users = await User.find();

res.status(200).json(users);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

router.put("/users/:id", async (req, res) => {

try {

const updatedUser = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });

if (!updatedUser) return res.status(404).json({ message: "User not found" });

res.status(200).json(updatedUser);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

router.delete("/users/:id", async (req, res) => {

try {

await User.findByIdAndDelete(req.params.id);

res.status(200).json({ message: "User deleted successfully" });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

module.exports = router;

**Screenshot:**

