- Databases
- Postgre and MySQL
- Mongodb Theory
    - Mongodb and SQL relation
- Basic Commands
    - Show databases
    - Creating and using the database
    - Insert data in collection
    - Retrieving the documents
        - Conditional retrieving
        - Projection
    - Updating the documents
    - Deleting documents
    - Aggregation
- Mongoose
    - Project Setup
- Files uploader (Express and Multy)
- Middle wear
- API Creation
- Socket.io
- Design Patterns

Databases

When considering databases and their types, it's important to understand that the "better" option for write, read, or management can vary depending on the specific requirements and characteristics of your application or use case. Here's a general overview:

1. Write Performance:

    - For high-volume write operations, NoSQL databases such as MongoDB, Cassandra, or Apache HBase are often preferred. They are designed to handle massive write scalability and offer high throughput for write-heavy workloads.
    - However, some relational databases like PostgreSQL and MySQL have also made significant improvements in their write performance and can handle

moderate to heavy write loads efficiently.

2. Read Performance:

    - Read performance can be influenced by factors such as data modeling, indexing strategies, and caching mechanisms.
    - Relational databases like PostgreSQL, MySQL, and Oracle are commonly used for read-intensive workloads, as they provide efficient indexing options, query optimization, and caching mechanisms to enhance read performance.
    - NoSQL databases, especially those optimized for read-heavy scenarios like Apache Cassandra or Elasticsearch, can also provide excellent read performance by distributing data across multiple nodes.

3. Management and Features:

    - Relational databases such as PostgreSQL, MySQL, and Oracle generally offer more mature and robust management tools and support. They have a long history and well-established ecosystems with a wide range of tools, libraries, and frameworks.
    - NoSQL databases often have simpler management requirements, as they may not require complex schemas or transactions. However, the management tools and support may not be as extensive as those available for relational databases.

It's essential to note that the suitability of a particular database for write, read, or management also depends on factors like data size, expected growth, data complexity, scalability requirements, and the expertise of your development team.

In some cases, a combination of databases, such as using a relational database for structured data and a NoSQL database for unstructured data or caching, might be appropriate. It's crucial to thoroughly analyze your specific use case and consider factors like data model, query patterns, scalability needs, and management requirements to determine the best-suited database or combination of databases for your application.

Postgre and MySQL

Both PostgreSQL (Postgres) and MySQL are popular relational database management systems (RDBMS) that have been widely used for many years. While both databases

have similar functionality and can handle read and write operations effectively, they do have some differences that can affect their performance in certain scenarios. Let's discuss them:

1. Write Performance:
    - MySQL traditionally had an advantage in write-intensive workloads, as it offered better performance for high-volume writes due to its simpler architecture and lighter transactional overhead.
    - However, PostgreSQL has made significant improvements in recent versions and has closed the gap in terms of write performance. It now provides comparable write speeds, especially for moderate to heavy write loads.
2. Read Performance:
    - Both databases are capable of handling read-intensive workloads effectively. The performance differences between them in read operations are generally negligible and highly dependent on the specific use case and configuration.
3. Management and Features:
    - PostgreSQL is known for its rich set of features, extensibility, and advanced data types. It supports a wide range of data types, including JSON, array types, and spatial data. PostgreSQL also offers more advanced functionalities like advanced indexing options, complex queries, and powerful built-in functions.
    - MySQL, on the other hand, has a simpler architecture and is generally considered easier to manage. It has a larger user base and a robust ecosystem of tools and libraries. MySQL is often favored in scenarios where ease of use and simplicity are important.
4. Data Integrity and Concurrency:
    - PostgreSQL has a strong emphasis on data integrity and provides advanced concurrency control mechanisms. It supports multi-version concurrency control (MVCC), which allows for high levels of concurrent read and write operations while maintaining data consistency.
    - MySQL traditionally used a simpler locking mechanism, but it has also introduced support for MVCC in recent versions, improving its concurrency capabilities.

In summary, both PostgreSQL and MySQL are powerful RDBMS options with their own strengths. PostgreSQL offers more advanced features and extensibility, making it suitable for complex data models and scenarios where data integrity and advanced functionality are crucial. MySQL, on the other hand, is often chosen for its ease of use, simplicity, and historical advantage in write-intensive workloads. However, it's important to note that performance and suitability can vary depending on specific use cases, dataset sizes, hardware configurations, and tuning optimizations.

Mongodb Theory

MongoDB is a popular open-source, document-oriented database management system. It falls under the category of NoSQL (non-relational) databases and is designed to handle large amounts of unstructured data. MongoDB provides high scalability, performance, and flexibility for managing and querying data.

Key features of MongoDB include:

1. Document-Oriented: MongoDB stores data in flexible, JSON-like documents called BSON (Binary JSON). Documents can have varying structures, allowing for easy representation of complex relationships and hierarchical data.

2. Scalability: MongoDB scales horizontally by sharding, which involves distributing data across multiple servers. This allows for the handling of large datasets and high traffic loads.

3. Flexible Data Model: MongoDB's flexible schema enables developers to iterate quickly and adapt to evolving data requirements. Fields can vary across documents, and data can be easily modified or extended.

4. High Performance: MongoDB utilizes memory-mapped storage and provides indexes to support efficient querying. It also supports automatic sharding and replication for improved performance and fault tolerance.

5. Rich Query Language: MongoDB supports a powerful query language with features like ad-hoc queries, indexing, sorting, and aggregation. It also provides support for geospatial queries, text search, and full-text search.

6. Replication and High Availability: MongoDB supports replica sets, which are self-healing clusters of MongoDB instances. Replica sets provide data redundancy, automatic failover, and high availability.

7. Distributed Transactions: MongoDB introduced multi-document ACID (Atomicity, Consistency, Isolation, Durability) transactions in version 4.0, allowing multiple operations to be grouped together in a single transaction.

8. Integration and Ecosystem: MongoDB provides drivers for various programming languages, making it easy to integrate with different frameworks and applications. It also integrates with popular analytics and visualization tools.

MongoDB is commonly used for a wide range of applications, including content management systems, real-time analytics, IoT (Internet of Things), social media platforms, and more.

Mongodb and SQL relation

- Table as Collection
- Row as Document

Basic Commands

Show databases

```
show dbs;
show databases;
```

Creating and using the database

If you want to create a new database then there should be at least one collection in it.

```
use bookstore;
db.createCollection("books");
show databases;
show collections;
```

Insert data in collection

```
db.createCollection("test");
db.test.find();
db.test.insertMany([{ name: "KD" }, { price: 123, product: "ABC" }]);

db.students.insertOne({
  name: "Ibrahim",
  age: 23,
  subjects: ["EAD", "PDC"],
  address: { city: "Hyderabad", country: "Pakistan" },
});
db.students.insertMany([
  {
    name: "John",
    age: 20,
    subjects: ["Math", "Science"],
    address: {
      city: "New York",
      country: "USA",
    },
  },
  {
    name: "Alice",
    age: 22,
    subjects: ["History", "English"],
    address: {
      city: "London",
      country: "UK",
    },
  },
  {
    name: "Bob",
    age: 19,
    subjects: ["Math", "Physics"],
    address: {
      city: "Paris",
      country: "France",
    },
  },
  {
    name: "Eva",
    age: 21,
    subjects: ["Biology", "Chemistry"],
    address: {
      city: "Berlin",
      country: "Germany",
    },
  },
  {
    name: "David",
    age: 20,
    subjects: ["Math", "English"],
```

```
    address: {
      city: "New York",
      country: "USA",
    },
  },
  {
    name: "Sarah",
    age: 23,
    subjects: ["Physics", "Chemistry"],
    address: {
      city: "London",
      country: "UK",
    },
  },
  {
    name: "Michael",
    age: 19,
    subjects: ["History", "Geography"],
    address: {
      city: "Paris",
      country: "France",
    },
  },
  {
    name: "Emily",
    age: 20,
    subjects: ["Biology", "English"],
    address: {
      city: "Berlin",
      country: "Germany",
    },
  },
  {
    name: "Daniel",
    age: 22,
    subjects: ["Math", "Chemistry"],
    address: {
      city: "New York",
      country: "USA",
    },
  },
  {
    name: "Sophia",
    age: 21,
    subjects: ["Physics", "English"],
    address: {
      city: "London",
      country: "UK",
    },
  },
]);
```

Retrieving the documents

```
db.students.find();
db.students.findOne();
db.students.find().count();

db.students.distinct("name");
```

Conditional retrieving

```
// Doc having age 20
db.students.findOne({ age: 20 });
// Doc having name = "Daniel" and age = 22
db.students.find({ age: 22, name: "Daniel" }); // Checks both conditions must be
true

db.students.find({ name: { $eq: "Sophia" } });

db.students.find({ age: 22, "address.city": "London" });
// Age > 20
db.students.find({ age: { $gt: 20 } });

// Age >= 20
b.students.find({ age: { $gte: 20 } });

// Age != 20
db.students.find({ age: { $ne: 20 } });

db.students.find({ $and: [{ age: { $gt: 20 } }, { age: { $lt: 22 } }] });

db.students.find({ $or: [{ name: "Daniel" }, { name: "Sophia" }] });

db.students.findOne({ $or: [{ name: "Daniel" }, { name: "Sophia" }] }).address
  .city;

db.students.find({ name: { $in: ["Sophia", "Daniel"] } });
```

Projection

```
db.students.find({}, { name: 1, age: 1, _id: 0 });

db.students.find({}, { _id: 0, subjects: 0 });
```

Updating the documents

```
db.students.updateOne(
  { _id: ObjectId("6494207101574b73fb117ddb") },
  { $set: { age: 30 } }
);

db.students.updateOne(
  { _id: ObjectId("6494207101574b73fb117ddb") },
  { $set: { address: { city: "Sukkur", country: "Pakistan" } } }
);
```

Deleting documents

```
db.students.deleteOne({ name: "Shahid" });

db.students.deleteMany({});
```

Aggregation

Mongoose

Project Setup

```
npm init
npm i mongoose express ejs
npm i --dev nodemon
```

Create a folder named `models` and add `product.js` file

```
// use `await mongoose.connect('mongodb://user:password@127.0.0.1:27017/test');` if
your database has auth enabled
const connectionString = "mongodb://127.0.0.1:27017/ead";

const mongoose = require("mongoose");
mongoose.connect(connectionString);

const Schema = mongoose.Schema;
```

```
const ProductSchema = new Schema({
  name: String,
  price: Number,
  qty: Number,
  manufacturer: String,
});

module.exports = mongoose.model("product", ProductSchema);
```

`index.js` code

```
const express = require("express");
const app = express();

const Product = require("./models/product");

app.use(express.json());
app.use(express.urlencoded());
app.set("view engine", "ejs");

app.get("/", (req, res) => {
  res.render("index");
});

app.get("/products", (req, res) => {
  Product.find().then((products) => {
    res.render("products", {
      title: "Products",
      products: products,
    });
  });
});

app.get("/product/new", (req, res) => {
  res.render("newProduct");
});

// Form data is in urlencoded form
app.post("/product/save", (req, res) => {
  const product = new Product(req.body);
  product
    .save()
    .then((product) => {
      if (!product) return res.redirect("/product/new");
      res.redirect("/products");
    })
    .catch((error) => {
      console.log(error);
      res.redirect("/product/new");
    });
});

app.listen(3000, () => {
```

```
  console.log("Server is running on port 3000.");
});
```

`newProduct.ejs` file

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>New Product</title>
  </head>

  <body>
    <header><%-include("navbar")%></header>
    <form action="/product/save" method="post">
      <label
        >Product Name:
        <input type="text" name="name" placeholder="Enter product name" />
      </label>
      <br />
      <br />

      <label
        >Price:
        <input type="number" name="price" placeholder="Enter product price" />
      </label>
      <br />
      <br />

      <label
        >Quantity:
        <input type="number" name="qty" placeholder="Enter product quantity" />
      </label>
      <br />
      <br />

      <label
        >Manufacturer:
        <input
          type="text"
          name="manufacturer"
          placeholder="Enter Manufacturer"
        />
      </label>
      <br />
      <br />

      <input type="submit" value="Save" />
    </form>
  </body>
</html>
```

products.ejs file

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <%-include("navbar")%>
        <h1>
            <%=title%>
        </h1>
        <table>
            <thead>
                <th>Id</th>
                <th>Name</th>
                <th>Price</th>
                <th>Quantity</th>
                <th>Manufacturer</th>
            </thead>
            <tbody>

                <% for(let i=0; i<products.length; i++) { %>
                    <tr>
                        <td>
                            <%= products[i].id %>
                        </td>
                        <td>
                            <%= products[i].name %>
                        </td>
                        <td>$<%= products[i].price %>
                        </td>
                        <td>
                            <%= products[i].qty %>
                        </td>
                        <td>
                            <%= products[i].manufacturer %>
                        </td>

                    </tr>
                    <% } %>



            </tbody>
        </table>

</body>

</html>
```

`navbar.ejs` file

```
<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/products">Products</a></li>
  <li><a href="/product/new">Add new Product</a></li>
</ul>
```

Files uploader (Express and Multy)

Middle wear

Intercept the request or response Request comes from client and operations are performed before the route

```
const validateProductMiddleWare = (req, res, next) => {
  const { name, price, qty, manufacturer } = req.body;

  if (!name || !price || !qty || !manufacturer)
    return res.redirect("/product/save");
  next();
};

// Form data is in urlencoded form
app.post("/product/save", validateProductMiddleWare, (req, res) => {
  const product = new Product(req.body);
  product
    .save()
    .then((product) => {
      if (!product) return res.redirect("/product/new");
      res.redirect("/products");
    })
    .catch((error) => {
      console.log(error);
      res.redirect("/product/new");
    });
});
```

API Creation

JWT (JSON web token) tokens are used for user authentication most of the times. It is a package. Tokens are expired after some time. Not everyone should access the api

CORS: Cross Origin that is allowed in header as domains

Normally browsers do not allow CORS.

How to use JWT and handle CORS?

```
app.get("/products/get", async (req, res) => {
  const products = await Product.find();
  res.json(products);
});
```

Socket.io

Two way communication Master(Client)-Slave(Server) Architecture in web

Every request has TTL (Time-To-Live). Short Polling and Long Polling

Pull Request and Push Request

Client pulls the data request Server pushes the data.

Client sends request, server responds normally! What is going on server, client does not know normally.

SSE Server Sent Event: Data from Server to Client Flow

Design Patterns