Databases for Web Development

Introduction to Databases

A database is a systematic collection of data. For example, A university database that stores information about students, courses, and faculty. This database systematically organizes data such as student names, course titles, grades, and faculty departments. **It supports electronic storage and manipulation of data**. For example, An online retail store's database. This database stores product listings, customer information, and transaction records. It allows for the electronic manipulation of this data, such as updating stock levels, adding new customer details, or modifying order statuses. **Databases make data management more efficient and secure.** For instance, A hospital's patient record system. This database efficiently manages large volumes of sensitive patient data, including medical histories, treatment plans, and personal information. It ensures data security through access controls and encryption, allowing only authorized personnel to view or modify patient records.

Why are Databases Important?

Data Persistence: Databases store data persistently, allowing for retrieval and modification over time. For instance, Student records are maintained over the years, allowing the university to access historical academic records, track alumni progress, and make data-driven decisions for institutional improvements.

Data Organization: They provide a structured way to organize large amounts of data. For instance, The store's database categorizes products by various attributes like price, category, and brand, making it easy to manage inventory, analyze sales trends, and enhance customer shopping experiences.

Data Integrity and Security: Databases enforce rules to ensure data integrity and provide secure access to data. For instance, The hospital's database ensures that patient records are accurate and consistent. It restricts access to sensitive patient information, complying with healthcare regulations like HIPAA, thereby maintaining confidentiality and trust.

Efficiency: Enhance the speed and efficiency of accessing data. For instance, When a student requests a transcript, the university's database allows for quick retrieval of the student’s academic history, streamlining administrative processes. Similarly, faculty can efficiently access course and student performance data for academic planning.

Where Do Databases Fit in the Big Picture of Full Stack Development?

Back-end Component

Databases are a crucial part of the back-end, handling data storage for the front-end interfaces. For example, for an E-commerce Website:

- **Front-end:** User interface where customers view products, add items to their cart, and make purchases.
- **Back-end Database:** Stores product information, user profiles, and order history. When a user views a product, the front-end requests this data from the back-end database to display it.

Interaction with Server-side Languages

Databases work in conjunction with server-side languages (like Node.js) and frameworks (like Express) to manage data. For Example, for a Social Media Platform

- **Server-side Language (Node.js) and Framework (Express):** Handle requests like user logins, posting updates, and fetching friend lists.
- **Database:** Stores user credentials, posts, and connections. Node.js interacts with the database to retrieve or update this information based on user actions.

APIs and Data Exchange: They play a pivotal role in data exchange between the server and client-side, often through APIs. For examples for an Online Booking System

- **APIs:** Facilitate communication between the front-end application (where users select travel dates and destinations) and the back end.
- **Database:** Stores information about available dates, pricing, and bookings. When a user searches for available dates, the API retrieves this data from the database to present options to the user.

Basic Concepts and Definitions

Data: Basic unit of information. For instance, In a customer database, an individual record containing a customer's name, address, and phone number represents a basic unit of data.

Database Management System (DBMS): Software to create and manage databases. For example the following are different DBMS software out there:

Schema: Defines the structure and organization of data. For example, In an employee database, the schema defines the structure, such as 'EmployeeID' as an integer, 'Name' as a string, 'Salary' as a decimal, etc.

Query: A request for data retrieval or manipulation. For instance, In a library database, a query might be used to find all books published after 2010 by a certain author.

Index: Optimizes data retrieval speeds. A student database might have an index on the 'StudentID' field to speed up the search and retrieval of student records.

Examples of DBMSs

1. **MySQL:**
    - A popular open-source relational database management system.
    - Commonly used for web applications and online publishing.
2. **PostgreSQL:**
    - An advanced open-source relational database.
    - Known for its robustness, scalability, and support for advanced data types and SQL standards.
3. **Microsoft SQL Server:**
    - A relational database management system developed by Microsoft.
    - Widely used in enterprise environments, especially those relying on other Microsoft technologies.
4. **MongoDB:**
    - A popular NoSQL database, particularly document-oriented.
    - Used for storing semi-structured data and known for its scalability and flexibility
5. **Cassandra:**
    - A free and open-source, distributed, wide column store, NoSQL database management system.
    - Designed to handle large amounts of data across many commodity servers.
6. **Redis:**
    - An in-memory data structure store, used as a database, cache, and message broker.
    - Known for its speed and use in caching scenarios.

Structured Query Language (SQL)

- SQL stands for **Structured Query Language**, which is a fancy way of saying "how to talk to databases".
- SQL lets you create, read, update and delete data from databases using simple commands like SELECT, INSERT, UPDATE and DELETE. Don't worry, they are not as scary as they sound.
- SQL is also very flexible and powerful. You can use it to filter, sort, join, aggregate and manipulate data in various ways to suit your needs.

NoSQL data bases

- **NoSQL Stands For:** "Not Only SQL."
- **Key Characteristics:**
  - Often does not use structured query language (SQL) for data manipulation.
  - May not ensure full ACID (Atomicity, Consistency, Isolation, Durability) properties.
  - Capable of handling large volumes of unstructured or semi-structured data.
- **Design Goals:**
  - Overcome limitations of traditional relational databases.
  - Suited for large-scale distributed data handling and high performance.
  - Offers flexible schema design.
- **Use Cases:** Particularly useful in big data applications and real-time web applications.
- **Types of NoSQL Databases:**
  - Document-oriented (e.g., MongoDB).
  - Key-value stores (e.g., Redis).
  - Wide-column stores (e.g., Cassandra).
  - Graph databases (e.g., Neo4j).

The Difference Between SQL and NoSQL Databases

SQL (Structured Query Language) Databases:

- **Relational:** Organize data into tables linked by relationships.
- **Schema-Strict:** Require predefined schema.
- **Examples:** MySQL, PostgreSQL.

NoSQL (Not Only SQL) Databases:

- **Non-relational:** Use a variety of data models, including document, key-value, wide-column, and graph.
- **Schema-Flexible:** No need for a predefined schema.
- **Scalability:** Often more easily scalable.
- **Examples:** MongoDB (document-based), Redis (key-value store).

Examples of SQL & NoSQL Databases

Category	Database Management Systems
SQL	MySQL
	PostgreSQL
	Microsoft SQL Server
	Oracle Database
	SQLite
NoSQL	MongoDB
	Cassandra
	Redis
	CouchDB

Common Database Methods (CRUD Operations)

CRUD is an acronym for Create, Read, Update, and Delete, which are the four basic functions of persistent storage in database management. Here's an overview of each operation with examples:

1. **Create:** Adds new data to the database. In a blog website's database, the 'Create' operation is used when a new blog post is written and needs to be added to the database.
2. **Read:** Retrieves data from the database. On an e-commerce site, when a customer views product details, the 'Read' operation is used to fetch and display information from the database.
3. **Update:** Modifies existing data in the database. In an employee management system, the 'Update' operation is used to change an employee's details, such as their job title or salary.

4. **Delete:** Removes data from the database. When a user decides to delete their account on a social media platform, the 'Delete' operation is used to remove their personal data from the database.

Steps to Connecting your Backend to a Database

Connecting a backend application to a database and starting to interact with it involves several key steps. These steps ensure that your application can effectively communicate with the database to perform CRUD (Create, Read, Update, Delete) operations. Here is an overview of the general process:

1. **Choose a Database Management System (DBMS):**
    - Select a DBMS that suits the needs of your application (e.g., MySQL, PostgreSQL for SQL databases or MongoDB, Cassandra for NoSQL databases).
2. **Install the Database Software:**
    - Install the chosen DBMS on your server or use a cloud-based service.
3. **Configure the Database:**
    - Set up the initial database schema (for SQL databases) or collections/documents (for NoSQL databases).
    - Configure necessary settings like user access, permissions, and security measures.
4. **Install Database Driver or ORM/ODM in Your Backend Application:**
    - For direct database interaction, install the appropriate database driver in your backend environment.
    - For an abstraction layer, use an ORM (Object-Relational Mapping) for SQL databases or ODM (Object Document Mapping) for NoSQL databases. Examples include Sequelize for SQL databases or **Mongoose for MongoDB**.
5. **Establish a Database Connection:**
    - Write code in your backend application to establish a connection with the database. This typically involves specifying the database URL, port, and credentials.
6. **Perform CRUD Operations:**
    - Implement functions in your backend to perform CRUD operations.
    - For SQL databases, this involves writing SQL queries.
    - For NoSQL databases, this involves using the database's API or query language.
7. **Handle Database Responses and Errors:**

- Write code to handle responses from the database after CRUD operations. This could include parsing data, handling success or failure, and managing exceptions or errors.

8. **Close the Database Connection:**
    - Ensure that your application properly closes the database connection when it is no longer needed, especially in applications that open and close connections frequently.

Introduction to MongoDB

What is MongoDB?

MongoDB is a popular NoSQL database that uses a document-oriented data model and is known for its high scalability and flexibility in handling diverse data types. Instead of tables and rows as in SQL, MongoDB uses collections and documents. Data is stored in a format similar to JSON (JavaScript Object Notation), making it intuitive for those familiar with JavaScript.

```
// A JSON-like document in MongoDB
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "name": "Alice Smith",
  "email": "alice@example.com",
  "tags": ["developer", "mongodb"],
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

How MongoDB Interacts with Node.js and Express

MongoDB with Node.js

Node.js can easily connect to MongoDB, allowing for real-time data processing and manipulation.

Use of Mongoose

Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js, which simplifies data validation, casting, and business logic.

Integrating with Express

MongoDB can be integrated with Express to create a complete back-end solution for web applications.

Hands-On

Create a **MongoDB's cloud (Atlas) database**

1. Go to the MongoDB Atlas website and sign up for an account.
2. Follow the prompts to create your account.
3. Once logged in, you'll be prompted to create a cluster.
4. Choose a cloud provider (AWS, Google Cloud, or Azure) and a region that's closest to your users.
5. Select a cluster tier. You can start with the free tier for experimentation.
6. Click "Create Cluster."
7. Navigate to the "Network Access" section in the security settings.
8. Add an IP address to allow connections to your cluster. You can allow access from anywhere or specify a particular IP address for enhanced security.
9. In the "Database Access" section, create a new database user.
10. Provide a username and a password. Remember these credentials, as you'll need them to connect to your database.
11. Once your cluster is ready, click the "Connect" button.
12. Choose how you want to connect to your database. For application connections, select "Connect your application."
13. Copy the provided connection string and save it

Paste it here:

Download MongoDB Compass GUI

MongoDB Compass is the official GUI for MongoDB, which allows you to visualize and manipulate your database data more easily.

1. Go to the MongoDB Compass download page.
2. Select the version of Compass that matches your operating system (Windows, macOS, Linux).
3. Click the download link to start the download of MongoDB Compass
4. Once downloaded, run the installer, and follow the installation prompts.

Connecting MongoDB Compass to Your Database

1. Open MongoDB Compass once installed.
2. If you're using MongoDB Atlas, log in to your Atlas account, go to your cluster and click on 'Connect'.
3. Choose 'Connect using MongoDB Compass' and copy the provided connection string.
4. In MongoDB Compass, paste your connection string into the textbox.
5. If your connection string includes a placeholder for a password (like **<password>**), replace it with your actual database password.
6. Click the 'Connect' button to establish the connection.

Connecting to MongoDB using Node.js

1. Open a new VSCode window
2. Click on Clone Git Repository
3. Clone and open: https://github.com/mongodb-university/atlas_starter_nodejs
4. Change the placeholder connection string to yours
5. Run `node app.js` in the command line
6. Go over the code and understand the different parts of the code
7. Go to https://www.mongodb.com/developer/products/mongodb/cheat-sheet/ to learn more about MongoDB

Connecting your app using mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and translates between objects in code and the representation of those objects in MongoDB.

1. Create a new Node.js project: Initialize a new project with `npm init -y`

2. Install Mongoose: Run `npm install mongoose` to add Mongoose to your project.
3. Create a file for your application, e.g., `app.js`.
4. Write the code to connect to MongoDB:

```
const mongoose = require('mongoose');
mongoose.connect('YOUR_CONNECTION_STRING', { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB...'))
  .catch(err => console.error('Could not connect to MongoDB...', err));
```

- `const mongoose = require('mongoose');:` This line imports Mongoose into your file.
- `mongoose.connect(YOUR_CONNECTION_STRING', {...options})`: This method establishes a connection to your MongoDB database. Replace `YOUR_CONNECTION_STRING` with your actual MongoDB URI, which you can get from MongoDB Atlas or your local MongoDB server.
- The options object `{ useNewUrlParser: true, useUnifiedTopology: true }` contains flags to use the new MongoDB driver's URL string parser and server discovery and monitoring engine, respectively.

5. Run your Node.js application: Execute `node app.js` in your terminal to run the application.

CRUD Operations Examples Using Mongoose

Using the same previous project try the following functions to practice CRUD operations using mongoose ODM

```
const mongoose = require('mongoose');


// connect to db
mongoose.connect('YOUR_STRING_HERE', {
      useNewUrlParser: true,
      useUnifiedTopology: true
   })
   .then(() => console.log('Connected to MongoDB...'))
   .catch(err => console.error('Could not connect to MongoDB...', err));
```

```
// define item
const itemSchema = new mongoose.Schema({
    name: String
});

const Item = mongoose.model('Item', itemSchema);

// Create
async function createItem(name) {
    try {
        const item = new Item({
            name
        });
        await item.save();
        console.log('Item created:', item);
        return item; // Returning the created item
    } catch (error) {
        console.error('Create operation failed:', error);
    }
}

// Read
async function readItem(id) {
    try {
        const item = await Item.findById(id);
        console.log('Item read:', item);
        return item; // Returning the found item
    } catch (error) {
        console.error('Read operation failed:', error);
    }
}

// Update
async function updateItem(id, newName) {
    try {
        const item = await Item.findByIdAndUpdate(id, {
            name: newName
```

```
        }, {
            new: true
        });
        console.log('Item updated:', item);
        return item; // Returning the updated item
    } catch (error) {
        console.error('Update operation failed:', error);
    }
}

// Delete
async function deleteItem(id) {
    try {
        await Item.findByIdAndDelete(id);
        console.log('Item deleted');
    } catch (error) {
        console.error('Delete operation failed:', error);
    }
}

// call funcs
async function performCRUDOperations() {
    try {
        // Create an item
        const createdItem = await createItem('Sample Item');

        // Read the item
        await readItem(createdItem._id);

        // Update the item
        await updateItem(createdItem._id, 'Updated Item');

        // Delete the item
        await deleteItem(createdItem._id);

    } catch (error) {
        console.error('Error in CRUD operations:', error);
```

```
    } finally {
        mongoose.connection.close();
        console.log('MongoDB connection closed');
    }
}

// run
performCRUDOperations();
```

Challenges

The following are some challenges that you should be able to know how to do if you understood the sections above.

1. Connect your current website to a MongoDB database
2. Add some manual data to your MongoDB Atlas DB
3. Read and display the data using Mongoose
4. Create a new collection in MongoDB for the form data
5. When someone submits any form data store it in the collection