Introduction To Databases

Software Development Bootcamp

Topic

Database Types

Databases

- **Database**: a structured set of data held in a computer, usually organized so that it can be easily accessed, managed and updated

Database Types: Relational

- Very similar to a table structure (like Excel or Google Sheets)
- Have a strict structure
- Tables can be linked using Keys
- Ex: SQL, MySQL, PostgresSQL

Database Types: Non-Relational

- Uses a Primary Key (ID) to give individuality to each document
- Data is stored as documents containing a JSON object
- Ex: MongoDB

Topic

Introduction To MongoDB

What Is MongoDB?

MongoDB is a popular, open-source NoSQL database system that provides high performance, high availability, and easy scalability. Key features include:

- **Document-oriented storage**: Data is stored in flexible, JSON-like documents
- **Flexibility:** You can add new types of information without messing up what's already there.
- **Scalability:** As your data gets bigger, MongoDB can spread it across many computers to handle it all.
- **JavaScript:** If you know JavaScript, you'll find it easier to work with MongoDB.

Key MongoDB Terms

- **Document**: Think of this as a single page in your digital filing cabinet. It holds all the information about one thing, like a user or a product.
- **Collection**: This is like a folder in your filing cabinet. It holds many related documents. For example, all user documents might go in a "users" collection.
- **Field**: This is a piece of information in a document. Like "name" or "age" for a user document.
- **Schema**: Think of this as a blueprint for your documents. It defines what fields a document should have and what type of data goes in each field
- **Model**: This is like a factory for documents. It uses the schema as a blueprint to create and manage documents in a specific collection.

MongoDB Compass

Compass is a Graphical User Interface (GUI) for working with MongoDB.

- Make sure you have MongoDB Compass installed on your computer.

Compass: Creating A New Collection

- Open MongoDB Compass
- Click “New connection +” (on your first use, a New Connection form will already be open)
- Click the “Connect” button (usually is green)
- This will take you to a new view where we can create or view databases
- Click the “+” button next to the “Databases” label
- Add a name for the Database and the Collection, then press the “create Database” button.
- You should now be able to see your new DB in the left sidebar
- **Mac Users:** You may need to run the command **brew services start mongodb-community** in order to connect.

Compass: Creating A New Document

- Click the “ADD DATA” dropdown and select “Insert Document”
- Add some JSON and click “insert”
- You should see your new document has been added to the DB
- Notice: An ID has been automatically created for the document.

MongoDB ObjectID

ObjectID is like a special name tag for each document in MongoDB.

- It's automatically created when you add a new document
- It's stored in the "_id" field of each document.
- The Unique ID is made up of a time stamp, a random number, an a counter that starts from that random number
- This special mix makes sure each ObjectId is unique, even if you're adding lots of documents at the same time.

Drivers

- MongoDB has its own query syntax that is similar to JavaScript.
- Programming languages use drivers to interact with MongoDB. We'll use MongoDB's Node.js driver.

Connecting to a Database

- npm install mongodb
- Once we have the driver installed, we can import the MongoClient class from it and use that to generate a connection object.

MongoDB Connection String

A MongoDB connection string is like an address that tells your application how to find and connect to your database.

- Example: **mongodb://localhost:27017/myapp**

Keeping your Connection Secure

- A new MongoClient instance needs a connection string which will contain your username and password.
- We need to hide our password in environment variables and reference them through `process.env`
- We also need to make sure we don't upload our .env to github
- Create a `.gitignore` file and add your `.env` file to it

Setting up the Server

- To set up Mongo on the server we will need to install all necessary packages, and then:
  - Set up the express server
  - Open a connection to the Database
  - Optionally create some helper methods to more easily access the database
  - Start setting up routes

Using the MongoDB Drivers

- To use our Mongo drivers we first need to import the MongoClient class from the mongodb package.
- Note that we do need to destructure MongoClient out of our mongodb package.

```
const {MongoClient} = require('mongodb');
```

Connecting to the Collection

- Set up your client variable in the root level of your server file so it's accessible from any other functions that need it.
- **dbConnect():** Function we create and use for establishing a connection to the database.
    - Connects to MongoDB using the provided connection string
    - Returns a reference to the specified collection

```
const DB_URL = 'mongodb://localhost:27017'
// new mongo client
const client = new MongoClient(DB_URL)
async function dbConnect() {
   // establish connection with the
database process
   await client.connect()
   // create a database or connect if one
exists
   const db = await client.db('myDatabase')
   // create a collection within our new
database
   const collection = await
db.collection('users')
   return collection
}
```

POST Example

- **insertOne():** Mongo Method used for adding a single document to a collection
- Takes an object representing the document to insert (in this example its the **req.body**)
- Automatically adds an **_id** field

```
// middleware to parse JSON bodies REMEMBER this
comes before your routes
app.use(express.json())
// POST route
app.post('/create', async(req, res) => {
    try {
        // connect to our database and collection
        const connect = await dbConnect()
        // push the contents of the body into our
collection as a document
        const newItem = await
connect.insertOne(req.body)
        // if successful send status and JSON object
with message, and newly created document
        res.status(201).json({
            message: `user created`,
            newItem
        })
        // if unsuccessful log the error to the
console
    } catch (error) {
        console.log(error)
    }
})
```

GET Example

- **.find()**: Used for retrieving multiple documents from a collection
  - Takes no arguments, it returns all documents
  - Can take a query object to filter results
- **.toArray():** Converts the result of the **.find()** method to an array

```
// GET route
app.get('/getusers', async(req, res) => {
    try {
        // connect to database and collection
        const connect = await dbConnect()
        // .find() method returns a cursor object. Use
.toArray() method to turn cursor into an object
        const userList = await
connect.find({}).toArray()
        // if successful send status and JSON object
containing the userList
        res.status(200).json({
            userList
        })
        // if unsuccessful log the error to the console
    } catch (error) {
        console.log(err)
    }
})
```

Exercise

Robot Warehouse