SD540 Server-Side Programming

Maharishi University of Management

Masters of Software Development

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

What is MongoDB?

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

Non relational DB, stores BSON documents.

Schemaless: Two documents don't have the same schema.

Download and Install MongoDB Community Server, with Compass.

Non-Relational Database

Relational databases are like spreadsheets, organize data in rows and columns within tables. The tables are linked through specific keys, allowing for complex queries and data integrity.

Non-relational databases (NoSQL) store data in various formats like documents, key-value pairs, or graphs, and don't rely on rigid table structures. This makes them ideal for handling large, unstructured datasets and scaling with ease.

Data Model

A **Database** consists of one or more collections. **Collections** are like buckets that contain documents. There is no relationship between collections. A **Document** represents a single record (object).

Values may contain other documents (embedded documents), arrays, and arrays of documents (**Rich Documents**).

```
{
   _id: 1,
   name: "Asaad",
   email: "asaad@miu.edu",
   courses: ["CS472", "CS572"]
}
```

Data Types

The value of a field can be any of the **BSON data types**.

```
const mydoc = {
        _id: ObjectId("5099803df3f4948bd2f98391"),
        name: { first: "Asaad", last: "Saad" },
        birth: new Date('Oct 31, 1979'),
        courses: [ "CS472", "CS572" ],
        students : NumberLong(1250000)
}
```

BSON Types: String, Integers, Double, Decimal128, Boolean, Null, Date, ObjectId, Array.. etc

https://www.mongodb.com/docs/manual/reference/bson-types/

Atomicity

MongoDB guarantees atomicity on the document level. Which means that any operation is either all applied or not applied.

General Rules

- The field name **`_id`** is reserved for use as a **primary key**. It is **immutable** and always the first field in the document. It may contain values of any other BSON data type, but not an array.
- The maximum BSON document size is 16 megabytes.

MongoDB Schema Design

In MongoDB we use **Application-Driven Schema**, which means we design our schema based on **how we access the data**.

Note: The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection.

Take into considerations the following: Frequently of access and the way you want to access the data. Size of items (16M). Atomicity of data.
Benefits of Embedding: Improved read performance. One round to the DB.

Object Modeling for Node

Mongoose ODM is an object document modeling package.

```
// install mongoose package
$ npm install mongoose

// import the library in your application
import mongoose from 'mongoose';

// connect to a MongoDB database
await mongoose.connect('mongodb://127.0.0.1:27017/databaseName');

// use mongoose.connect('mongodb://user:password@127.0.0.1:27017/db'); if your database has auth enabled
```

Mongoose Schema and model

```
import { InferSchemaType, Schema, model } from 'mongoose';

// create a schema
const schema = new Schema(({
    fullname: String,
    username: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    active: { type: Boolean, default: true },
    height: Number,
    weight: Number,
    gender: { type: String, enum : ['male','female'], default: 'male'}
}, { timestamps: true });

// create an inferred type from your schema
export type User = InferSchemaType<typeof schema>;

// create a model object using schema
const userModel = model<User>('user', schema)
```

```
SchemaTypes
 String
 Number
 Date
 Buffer
 Boolean
 Mixed
 ObjectId
 Array
```

http://mongoosejs.com/docs/guide.html

Create New Document

```
const theo: User = await UserModel.create({ username: 'theo', password: '123456' });

const users: User[] = await UserModel.create([
     { username: 'mike', password: '123456' },
     { username: 'mada', password: '123456' }
]);
```

`find({query}, {projection})`

Selects documents in a collection and returns an array of the results.

```
// returns array of all users
const users: User[] = await UserModel.find({});

// returns array of all passwords
const users: User[] = await UserModel.find({}, {_id: 0, password: 1});

// returns array of all admins
const users: User[] = await UserModel.find({admin: true});
```

The projection argument cannot mix include and exclude specifications, except for the **`_id`** field.

findOne({query}, {projection})

Returns the **first document** that satisfies the specified query criteria. If no document satisfies the query, the method returns null.

```
// returns the first user
const firstuser: User | null = await UserModel.findOne({});

// returns theo's password and _id
const theo: User | null = await UserModel.findOne({ username: 'theo' }, { password: 1 });
```

Query Operators

Can be applied to **numeric** and **string** field values

`$gt` greater than

`$gte` greater than or equal to

`$lt` less than

`$lte` less than or equal to

`$in` matches any of the values specified in an array (implicit OR)

`Comma` between operators works as (implicit AND)

```
{field: {operator: value} }
```

Examples - Query Operators

```
// return all users with weight more than 85
const users: User[] = await UserModel.find({ weight: { $gt: 85} })

// return all users with a weight is either 5 or 15 (implicit OR)
const users: User[] = await UserModel.find({ weight: { $in: [5, 15] } })

// return all users with weight lower than 85 AND height is 6 (implicit AND)
const users: User[] = await UserModel.find({ weight: { $lt: 85 }, height: 6 })
```

Logical Query Operators

Joins query clauses with a logical operation:

$or returns all documents that match the conditions of **either** clause.

$and returns all documents that match the conditions of **both** clauses.

```
{ $or:  [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
{ $and: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
```

```
// return all users where either the weight is less than 20 or the height is 10
const users: User[] = await UserModel.find(
          { $or: [
               { weight: { $lt: 20 } },
               { height: 10 }
           ] } )
```

Implicit vs. Explicit Logical Operators

Use the implicit logical operators when you need to apply them on the same field and use the explicit logical operators when you need to apply them on different fields.

Sort the Results

```
// sort by username in ascending order
const users: User[] = await UserModel.find({}).sort({ username: 1 });

// sort by username in descending order
const users: User[] = await UserModel.find({}).sort({ username: -1 });

// sort by active in ascending order, within each group, sort by fullname in asc
const users: User[] = await UserModel.find({}).sort({ active: 1 , fullname: 1 });
```

Results Pagination

```
const users: User[] = await UserModel.find({})
                                     .skip((page -1) * pageSize)
                                     .limit(pageSize)
```

Note: Projection and Pagination are good practices to save bandwidth and retrieve only the data we need.

Delete Documents

```
// delete a document with username theo documents
const results = await UserModel.deleteOne({ username: 'theo' }) // {deletedCount: 1}

// delete all documents
const results = await UserModel.deleteMany({}) // {deletedCount: x}
```

Update Documents

```
// Update only the first document that matches query.
const results = await UserModel.updateOne(
                                         { username: 'theo' },
                                         { $set: { weight: 17.6 } })

results.matchedCount;  Number of matched documents
results.modifiedCount;  Number of modified documents
results.acknowledged;  Boolean indicating everything went smoothly.
results.upsertedId;  null or an id containing a document that had to be upserted.
results.upsertedCount;  Number indicating how many documents had to be upserted.
```

If you don't include any update operators, Mongoose will wrap the update part in `$set` for you. This prevents you from accidentally overwriting the document.

Upsert Documents

```
// Update if found, insert if not found.
const results = await UserModel.updateOne(
                                  { username: 'theo' },
                                  { $set: { weight: 17.6 } },
                                  { upsert: true });
```

Update Operators

```
// add 2 pounds to all users' weight
const results = await UserModel.updateMany(
                                  { },
                                  { $inc: { weight: 2 } })

// decrease 2 pounds from theo's weight
const results = await UserModel.updateOne(
                                  { username: 'theo' },
                                  { $inc: { weight: -2 } })
```

Handling Large Multi-updates

When we do multi-documents write operations inside MongoDB, there is a **single thread for every operation being executed** (run sequentially inside single thread). Every write operation that affects more than one document is carefully coded in multi-tasking way to occasionally **yield control** to allow other operations to work on the same dataset.

MongoDB guarantees an **individual** doc to be **Atomic**, so no other R/W operation can get a half-updated document.

Multi-doc update operations are **not isolated transactions**.