1. Data Modeling and Schema Design in MongoDB

MongoDB's flexible schema allows dynamic document structures. However, effective schema design is crucial for performance, scalability, and maintenance.

Core Concepts in MongoDB Schema Design

1. **Documents and Collections**:
    a. A **document** is a JSON-like object stored in a MongoDB collection.
    b. A **collection** is a group of related documents.
2. **Field Types**: MongoDB supports various data types:
    a. `String`, `Number`, `Array`, `Boolean`, `Date`, `ObjectId`, `Embedded Document`, etc.
3. **Embedding vs. Referencing**:
    a. **Embedding**: Nesting documents within another document.
    b. **Referencing**: Storing references to related documents.

Embedding vs. Referencing - Examples

Embedding Example

This approach is used when related data is small and frequently queried together.

Use Case: A blog post with embedded comments.

```
json
Copy code
{
  "title": "Learn MongoDB",
  "author": "John Doe",
  "comments": [
    { "username": "Alice", "text": "Great post!", "createdAt": "2024-01-01" },
    { "username": "Bob", "text": "Very helpful!", "createdAt": "2024-01-02" }
  ]
}
```

Pros:

- Data is stored together and easy to query.
- Reduces need for multiple queries.

Cons:

- Size of the document increases as comments grow.

Referencing Example

This approach is used when related data grows large or requires frequent updates.

Use Case: Separate posts and comments collections.

Posts Collection:

json
Copy code

```
{
  "_id": 1,
  "title": "Learn MongoDB",
  "author": "John Doe"
}
```

Comments Collection:

json
Copy code

```
{
  "post_id": 1,
  "username": "Alice",
  "text": "Great post!",
  "createdAt": "2024-01-01"
}
```

Pros:

- Better for large and frequently updated datasets.
- Reduces duplication of data.

Cons:

- Requires JOIN-like operations to fetch related data.

When to Embed vs. Reference

- Use **Embedding** when:
  - Data is accessed together (e.g., comments on a blog post).
  - Data size is small.
- Use **Referencing** when:
  - Data grows large or is frequently updated.
  - Data is shared across multiple collections.

2. Advanced Queries and Filters in MongoDB

MongoDB provides powerful query operators to filter and manipulate data.

Basic Query Examples

Equality:

Find documents where name is "Alice":

```
javascript
Copy code
db.users.find({ name: "Alice" });
```

Comparison Operators:

- `$gt`, `$gte`, `$lt`, `$lte` (greater/less than):

```
javascript
Copy code
db.users.find({ age: { $gt: 25 } });
```

Logical Operators:

- `$or` to match multiple conditions:

```
javascript
Copy code
db.users.find({
  $or: [{ city: "New York" }, { age: { $lte: 30 } }]
});
```

Array Queries

1. **Match exact array**:

```
javascript
Copy code
db.posts.find({ tags: ["mongodb", "database"] });
```

2. **Match element in an array**:

```
javascript
Copy code
db.posts.find({ tags: { $in: ["mongodb"] } });
```

Aggregation Framework

The **aggregation framework** processes documents and transforms them into aggregated results.

Example: Count Users by City

```
javascript
Copy code
db.users.aggregate([
  { $group: { _id: "$city", userCount: { $sum: 1 } } }
]);
```

Example: Filtering and Grouping

Find users over 25 and group them by city:

```
javascript
Copy code
db.users.aggregate([
  { $match: { age: { $gt: 25 } } },
  { $group: { _id: "$city", count: { $sum: 1 } } }
]);
```

3. Introduction to Mongoose (ODM for MongoDB)

What is Mongoose?

- Mongoose is an **Object Data Modeling (ODM)** library for MongoDB and Node.js.
- It allows defining schemas, performing validations, and using powerful queries.

Setting Up Mongoose

1. **Install Mongoose**:

```
bash
Copy code
npm install mongoose
```

2. **Connect to MongoDB**:

```
javascript
Copy code
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/myapp', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

mongoose.connection.once('open', () => {
  console.log('Connected to MongoDB');
});
```

Define a Schema and Model

A **schema** defines the structure of the documents in a collection.

```
javascript
Copy code
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: { type: Number, required: true },
  city: String,
  createdAt: { type: Date, default: Date.now }
});

const User = mongoose.model('User', userSchema);
```

CRUD Operations with Mongoose

1. Create a Document

```
javascript
Copy code
const user = new User({ name: "Alice", age: 25, city: "New York" });

user.save()
  .then(doc => console.log("User Created:", doc))
  .catch(err => console.error(err));
```

2. Read Documents

Find all users:

```
javascript
Copy code
User.find().then(users => console.log(users));
```

Find one user:

```
javascript
```

```
Copy code
User.findOne({ name: "Alice" }).then(user => console.log(user));
```

Find users with a condition:

```
javascript
Copy code
User.find({ age: { $gt: 20 } });
```

3. Update Documents

Update a user:

```
javascript
Copy code
User.findOneAndUpdate({ name: "Alice" }, { city: "Los Angeles" },
{ new: true })
  .then(updatedUser => console.log("Updated User:", updatedUser));
```

4. Delete Documents

Delete one document:

```
javascript
Copy code
User.deleteOne({ name: "Alice" })
  .then(() => console.log("User Deleted"));
```

4. Complete Example: REST API with Mongoose

Create a simple REST API for user management using **Node.js**, **Express**, and **Mongoose**.

Step 1: Setup Project

1. **Install dependencies**:

```
bash
Copy code
npm init -y
npm install express mongoose body-parser
```

2. **Create `app.js`**:

```
javascript
Copy code
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/myapp', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

// Define Schema
const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
  city: String
});
```

```
const User = mongoose.model('User', userSchema);

// Routes
app.post('/users', (req, res) => {
  const user = new User(req.body);
  user.save().then(doc => res.send(doc));
});

app.get('/users', (req, res) => {
  User.find().then(users => res.send(users));
});

app.put('/users/:id', (req, res) => {
  User.findByIdAndUpdate(req.params.id, req.body, { new: true })
    .then(updatedUser => res.send(updatedUser));
});

app.delete('/users/:id', (req, res) => {
  User.findByIdAndDelete(req.params.id).then(() => res.send("User
Deleted"));
});

// Start Server
app.listen(3000, () => console.log("Server running on port 3000"));
```

3. **Run the API**:

```
bash
Copy code
node app.js
```

4. Test using Postman or a browser:
    a. POST `/users` → Add a user
    b. GET `/users` → Get all users
    c. PUT `/users/:id` → Update a user
    d. DELETE `/users/:id` → Delete a user

Summary

1. **MongoDB Data Modeling**:
    a. Embedding vs. Referencing.
2. **Advanced Queries**:
    a. `$gt`, `$lt`, `$or`, Aggregations.
3. **Mongoose**:
    a. ODM for MongoDB with schema, models, and CRUD operations.
4. **Project Example**:
    a. A REST API to practice MongoDB and Mongoose.