MongoDB

Different ways to connect Express App with MongoDB using mongoose

```
mongoose.connect('mongodb://localhost:27017/myUsers' , {
    useNewUrlParser :true,
    useUnifiedTopology: true,

}).then(() => {
    console.log("Express to MongoDB connection seccessfully...")
}).catch((error) => {
    console.log(error+" conection failed some internal issue !!!");
})
```

```
mongoose.connect('mongodb://username:password@localhost:27017/mydatabase',
{
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
.then(() => console.log('MongoDB Connected with Authentication'))
.catch(err => console.error('Connection error:', err));
```

Different Ways to Connect an Express App with MongoDB

When building a Node.js + Express application with MongoDB as the database, you have multiple ways to establish the connection. Each approach has its own use case, flexibility, and complexity. Let's go through them in detail.

✅ 1. Using Mongoose (Most Common Way)

Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js. It provides schema-based models, making it easier to interact with MongoDB.

📌 Installation

`npm install mongoose`

📌 Connection

```
const express = require("express");
const mongoose = require("mongoose");
const app = express();
const MONGO_URI = "mongodb://localhost:27017/mydatabase";  // Connect to
MongoDB

mongoose.connect(MONGO_URI, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
 });
const db = mongoose.connection;
db.on("error", console.error.bind(console, "Connection error:"));
db.once("open", () => console.log("Connected to MongoDB"));

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

✅ Key Features

- Schema-based models with validation.
- Built-in methods for querying and updating data.
- Middleware support for hooks like `pre` and `post` operations.
- Easy to implement relationships and populate queries.

✅ 2. Using MongoDB Native Driver

The MongoDB native driver provides direct access to MongoDB without any abstraction layer. It is efficient but requires more manual handling of schema validation.

📌 Installation

```
npm install mongodb
```

📌 Connection

```
const express = require("express");
const { MongoClient } = require("mongodb");
const app = express();
const MONGO_URI = "mongodb://localhost:27017";

const DB_NAME = "mydatabase";

async function connectDB() {
    const client = new MongoClient(MONGO_URI, {
         useNewUrlParser: true,
         useUnifiedTopology: true,
    });
    try {
         await client.connect();
            console.log("Connected to MongoDB");
            const db = client.db(DB_NAME);
            app.locals.db = db;  // Store the db instance in app.locals
for reuse

    }
    catch (err) {     console.error(err);   } }  connectDB();
```

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

✅ Key Features

- Direct MongoDB connection without an ODM.
- Fast and lightweight.
- More control over raw MongoDB operations.
- Ideal for performance-intensive apps or when you want to avoid the overhead of an ORM/ODM.

✅ 3. Using MongoDB Atlas (Cloud Database)

MongoDB Atlas is a cloud-based MongoDB service. You can connect your Express app to Atlas using either Mongoose or the MongoDB native driver.

📌 Steps to Use MongoDB Atlas

1. Create an account on MongoDB Atlas.
2. Create a cluster.
3. Get the connection string (e.g., `mongodb+srv://username:password@cluster0.mongodb.net/mydatabase`).
4. Use Mongoose or MongoDB native driver to connect.

📌 Connection with Mongoose

```
const express = require("express"); const mongoose = require("mongoose");
const app = express(); const MONGO_URI = "mongodb+srv://<username>:
<password>@cluster0.mongodb.net/mydatabase"; mongoose.connect(MONGO_URI, {
useNewUrlParser: true, useUnifiedTopology: true, });
mongoose.connection.on("connected", () => { console.log("Connected to MongoDB
Atlas"); }); app.listen(3000, () => console.log("Server running on port
3000"));
```

✅ Key Features

- Cloud-based, no need to set up MongoDB locally.
- Highly available and scalable.
- Built-in backup and monitoring.
- Suitable for production apps.

✅ 4. Using Mongoose with Connection Pooling

Connection pooling helps in reusing existing database connections, improving performance and efficiency.

📌 Connection with Pooling

```
const express = require("express"); const mongoose = require("mongoose");
const app = express(); const MONGO_URI =
"mongodb://localhost:27017/mydatabase"; // Use connection pooling const
options = { useNewUrlParser: true, useUnifiedTopology: true, poolSize: 10, //
Number of concurrent connections }; mongoose.connect(MONGO_URI, options);
const db = mongoose.connection; db.on("error", console.error.bind(console,
"Connection error:")); db.once("open", () => console.log("Connected with
```

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

✅ Key Features

- Improves performance by reusing connections.
- Reduces the overhead of creating new connections.
- Ideal for high-traffic apps.

✅ 5. Using `dotenv` for Configuration Management

Instead of hardcoding MongoDB connection strings, use `dotenv` to store environment variables securely.

📌 Installation

`npm install dotenv`

📌 `.env` File

`MONGO_URI=mongodb://localhost:27017/mydatabase PORT=3000`

📌 Connection

`const express = require("express"); const mongoose = require("mongoose"); const dotenv = require("dotenv"); dotenv.config(); const app = express(); mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true, }); mongoose.connection.on("connected", () => {`

```
console.log("Connected to MongoDB"); }); app.listen(process.env.PORT, () =>
console.log(`Server running on port ${process.env.PORT}`));
```

✅ Key Features

- Keeps sensitive information hidden.
- Easier configuration management.

✅ 6. Using `async/await` for Cleaner Connection

You can use `async/await` for cleaner and more readable code.

📌 Connection

```
const express = require("express"); const mongoose = require("mongoose");
const app = express(); const MONGO_URI =
"mongodb://localhost:27017/mydatabase"; const connectDB = async () => { try {
await mongoose.connect(MONGO_URI, { useNewUrlParser: true,
useUnifiedTopology: true, }); console.log("Connected to MongoDB"); } catch
(error) { console.error("Connection error:", error); } }; connectDB();
app.listen(3000, () => console.log("Server running on port 3000"));
```

✅ Key Features

- Clean and readable code.
- Better error handling with `try-catch`.

✅ 7. Using ORM Libraries (TypeORM, Prisma)

If you prefer using an ORM for MongoDB, you can use TypeORM or Prisma.

📌 TypeORM with MongoDB

`npm install typeorm mongodb`

`import "reflect-metadata"; import { createConnection } from "typeorm"; createConnection({ type: "mongodb", url: "mongodb://localhost:27017/mydatabase", useNewUrlParser: true, useUnifiedTopology: true, }) .then(() => console.log("Connected to MongoDB with TypeORM")) .catch((error) => console.log(error));`

✅ Key Features

- TypeORM provides decorators and entity-based schema management.
- Prisma offers modern and efficient ORM features.

Using `mongoose` with Connection Pooling

Connection pooling improves performance by reusing database connections.

Implementation

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

mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  poolSize: 5,  // Maximum connections
  serverSelectionTimeoutMS: 5000, // Timeout after 5 seconds
})
.then(() => console.log('MongoDB Connected with Connection Pooling'))
.catch(err => console.error('Connection error:', err));
```