



ix

Embrace Opportunity

Software Engineering Day 8

Express and MongoDB

iX



Today's Overview

- Introduction to MongoDB.
 - Connecting MongoDB with Node.js.
 - DB Models and CRUD Operations with Express.
 - Database Design Fundamentals
 - UML Diagrams.
-
- Cheat Sheet:
 - [iX Cheat Sheet](#)
 - [iX Cheat Sheet - Day 8](#)

MongoDB

An introduction to MongoDB.

MongoDB - Introduction

MongoDB is a open-source, cross-platform, document-oriented database. It stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time. As a NoSQL database, MongoDB is part of the modern web stack, particularly popular in building MEAN (MongoDB, Express.js, AngularJS, and Node.js) and MERN (MongoDB, Express.js, React.js, and Node.js) applications.

MongoDB - Key Features

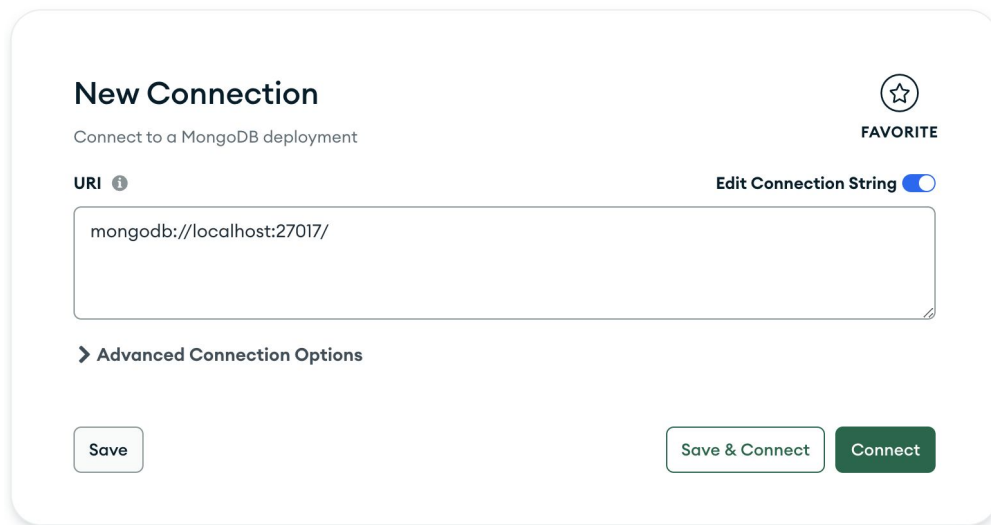
- **Document-Oriented Storage:** Data is stored in documents (similar to JSON objects) in a free schema design that allows you to easily incorporate changes.
- **High Performance:** Supports embedded data models to reduce I/O activity on the database server.
- **High Availability:** Its replication facility, called Replica Sets, provides automatic failover and data redundancy.
- **Scalability:** Easy horizontal scalability with sharding allows you to distribute data across multiple machines.

Install MongoDB


- [Install MongoDB Server Community Edition](#)
 - This installs the latest version of the MongoDB server on your local machine.
- [Download and Install MongoDB compass](#)
 - This GUI allows you to connect and interact with your local DB server.

MongoDB Compass

- Open MongoDB Compass and connect to local MongoDB server using the default connection string:
 - **mongodb://localhost:27017**



The screenshot shows the 'New Connection' dialog in MongoDB Compass. At the top, it says 'New Connection' with a star icon and the word 'FAVORITE'. Below this is the instruction 'Connect to a MongoDB deployment'. The 'URI' field is highlighted with an information icon, and the 'Edit Connection String' toggle is turned on. The text 'mongodb://localhost:27017/' is entered in the URI field. Below the field is a link for 'Advanced Connection Options'. At the bottom, there are three buttons: 'Save', 'Save & Connect', and 'Connect'.

New Connection  **FAVORITE**

Connect to a MongoDB deployment

URI ⓘ **Edit Connection String** ☒

mongodb://localhost:27017/

➤ **Advanced Connection Options**

Save **Save & Connect** **Connect**

MongoDB and Express JS

Connecting MongoDB to ExpressJS.

Install mongoose

- Install Mongoose npm package:

```
npm install mongoose
```

sh

Mongoose is a Object Data Modeling (ODM) library for MongoDB and Node.js. Mongoose is effectively an API that allows node application to interact with MongoDB. It provides a straight-forward, schema-based solution to model your application data. Mongoose offers built-in type casting, validation, query building, business logic hooks and more, out of the box.

Install dotenv

- Install Mongoose npm package:

```
npm install dotenv
```

sh

- Create `.env` in the `/backend/` directory and add the `MONGO_URI` environment variable and set it to your MongoDB connection string.

```
backend > ⚙ .env
```

```
1  PORT=5000
```

```
2  MONGO_URI=mongodb://localhost:27017/blog_app
```

```
3  
```

Install dotenv

- Import dotenv into project entry file, *src/index.js*:

```
const express = require("express");
require("dotenv").config();

const app = express();
const port = process.env.PORT || 8000;

app.use(express.json());

app.use("/api/blogs", require("../routes/blogs"));

app.listen(port, () => {
  console.log(`iX blogging app listening on port ${port}`);
});
```

Setup database connection in express

- In *backend/src/* create a database directory and add a file called *db.js*

```
✓ IX-BLOG-APP
  ✓ backend
    > node_modules
    ✓ src
      > controllers
      ✓ database
        JS db.js
      > routes
        JS index.js
      ⚙ .env
      {} package-lock.json
      {} package.json
```

Setup database connection in express

- Import mongoose & connect to db using connection string in *src/database/db.js*:

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

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log(`MongoDB Connected: ${conn.connection.host}`);
  } catch (error) {
    console.error(`Error: ${error.message}`);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Setup database connection in express

- Import and execute connectDB in project entry file, *src/index.js*:

```
const express = require("express");
const connectDB = require("../database/db");
require("dotenv").config();

connectDB();

const app = express();
const port = process.env.PORT || 8000;

app.use(express.json());

app.use("/api/blogs", require("../routes/blogs"));

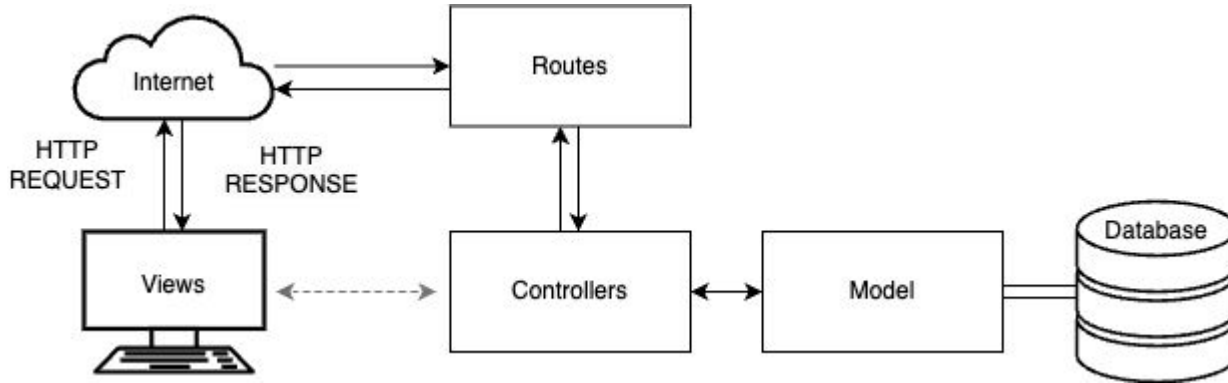
app.listen(port, () => {
  console.log(`IX blogging app listening on port ${port}`);
});
```

Models

Connecting MongoDB to ExpressJS.

MVC

- MVC or Model View Controller, is a **software architectural pattern** used for developing user interfaces. It divides the application into three interconnected components, allowing for efficient code organization and separation of concerns.



**Some server-side rendered frameworks like nextJS and Laravel the controller returns the populated view.*

Setup models in express

- Models are the Model part of the MVC software architectural pattern.
- In *backend/src/* create a models directory and add a file called *blogModel.js*

```

  ✓ backend
    > node_modules
    ✓ src
      > controllers
      > database
      ✓ models
        JS blogModel.js
      > routes
        JS index.js
    ⚙ .env
    {} package-lock.json
    {} package.json
```

Setup models in express

- Define the blog mongo DB schema (blog model) in *src/models/blogModel.js*:
- Define all fields and field types for blog model.
- Export an instance of mongoose model named “Blog” and schema definition.

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

const blogSchema = new mongoose.Schema(
  {
    authorId: {
      type: String,
      required: true,
    },
    categoryId: {
      type: String,
      required: true,
    },
    title: {
      type: String,
      required: true,
    },
    description: {
      type: String,
      required: true,
    },
    image: {
      type: String,
      required: true,
    },
    content: {
      type: Array,
      required: true,
    },
  },
  { timeStamp: true }
);

module.exports = mongoose.model("Blog", blogSchema);
```

Update blogs controller

- Import Blog model into, *src/controllers/blogs.js*

```
const Blog = require("../models/blogModel");
```

js

- Update the createBlog function in the blogs controller to use the Blog model.
 - createBlog function is converted to async since await blog.save(); is asynchronous.
 - Try catch error handling.
 - req.body cast to Blog.

Update blogs controller

```
const createBlog = async (req, res) => {  
  try {  
    const blog = new Blog({  
      authorId: req.body.authorId,  
      categoryId: req.body.categoryId,  
      readTime: req.body.readTime,  
      title: req.body.title,  
      description: req.body.description,  
      image: req.body.image,  
      content: req.body.content,  
    });  
    const newBlog = await blog.save();  
    res.status(201).json({ message: "New blog created!", data: newBlog });  
  } catch (error) {  
    res.status(500).json({ message: error.message, data: [] });  
  }  
};
```

js

Update blogs controller

```
const getBlogs = async (req, res) => {  
  try {  
    const blogs = await Blog.find();  
    res.status(200).json({ message: "Return all blogs!", data: blogs });  
  } catch (error) {  
    res.status(500).json({ message: error.message, data: [] });  
  }  
};  
  
const getBlog = async (req, res) => {  
  try {  
    const blog = await Blog.findById(req.params.id);  
    if (blog) {  
      res.status(200).json({ message: "Return blog by ID!", data: blog });  
    } else {  
      res.status(404).json({ message: "Blog not found!", data: [] });  
    }  
  } catch (error) {  
    res.status(500).json({ message: error.message, data: [] });  
  }  
};
```

js

Update blogs controller

```
const updateBlog = async (req, res) => {  
  try {  
    const blog = await Blog.findById(req.params.id);  
    if (blog) {  
      blog.authorId = req.body.authorId || blog.authorId;  
      blog.categoryId = req.body.categoryId || blog.categoryId;  
      blog.title = req.body.title || blog.title;  
      blog.description = req.body.description || blog.description;  
      blog.image = req.body.image || blog.image;  
      blog.content = req.body.content || blog.content;  
      const updatedBlog = await blog.save();  
      res.status(200).json({ message: "Blog updated!", data: updatedBlog });  
    } else {  
      res.status(404).json({ message: "Blog not found!", data: [] });  
    }  
  } catch (error) {  
    res.status(500).json({ message: error.message, data: [] });  
  }  
};
```

js

Update blogs controller

```
const deleteBlog = async (req, res) => {  
  try {  
    const blog = await Blog.findByIdAndRemove(req.params.id);  
    if (blog) {  
      return res.status(200).json({ message: "Blog deleted!" });  
    } else {  
      return res.status(404).json({ message: "Blog not found!" });  
    }  
  } catch (error) {  
    return res.status(500).json({ message: error.message });  
  }  
};
```

js

Database Design Fundamentals

Introduction to Database Design

Introduction

Certain principles guide the database design process.

1. **Duplicate information** (also called redundant data) **is bad**, because it wastes space and increases the likelihood of errors and inconsistencies.
2. **Correctness** and **completeness** of information is **important**. If your database contains incorrect information, any reports that pull information from the database will also contain incorrect information. As a result, any decisions you make that are based on those reports will then be misinformed.

Introduction

A **good database design** is, therefore, one that:

- **Divides information** into subject-based tables to reduce redundant data.
- Provides access with the information it requires to join the information in the tables together as needed.
- Helps **support** and **ensure** the **accuracy** and integrity of your information.
- **Accommodates** your **data processing** and reporting needs.

Introduction

The database design process consists of the following steps:

- **Determine** the **purpose** of your database
- **Find** and organize the **information** required
- **Divide** the information **into tables**
- Turn information **items into columns**
- **Specify** primary **keys**
- Set up the **table relationships**
- **Refine** your design

Determine the purpose of your database

It is a good idea to write down the purpose of the database on paper — its purpose, how you expect to use it, and who will use it. For a small database for a home based business, for example, you might write something simple like "The customer database keeps a list of customer information for the purpose of producing mailings and reports."

If the database is more complex or is used by many people, as often occurs in a corporate setting, the purpose could easily be a paragraph or more and should include when and how each person will use the database.

The idea is to have a well developed mission statement that can be referred to throughout the design process. Having such a statement helps you focus on your goals when you make decisions.

Finding and organizing the required information

To find and organize the information required, start with your existing information. For example, you might record purchase orders in a ledger or keep customer information on paper forms in a file cabinet. Gather those documents and list each type of information shown (for example, each box that you fill in on a form).

If you don't have any existing forms, imagine instead that you have to design a form to record the customer information.

For example, suppose you currently keep the customer list on index cards. Examining these cards might show that each card holds a customers name, address, city, state, postal code and telephone number. Each of these items represents a potential column in a table.

Dividing the information into tables

To divide the information into tables, choose the major entities, or subjects. For example, after finding and organizing information for a product sales database, the preliminary list might look like this:

```
Customers: {  
  name;  
  address;  
  city;  
  state;  
  mobileNumber;  
  emailAddress;  
}
```

```
Suppliers: {  
  name;  
  address;  
  city;  
  state;  
  mobileNumber;  
  emailAddress;  
}
```

```
Products: {  
  name;  
  description;  
  price;  
  unitsInStock;  
}
```

```
Orders: {  
  orderNumber;  
  salesPerson;  
  date;  
  orderItems;  
  total;  
}
```

Turning information items into columns

To determine the columns in a table, decide what information you need to track about the subject recorded in the table. For example, for the Customers table, Name, Address, City, E-mail address and Telephone number comprise a good starting list of columns. Each record in the table contains the same set of columns, so you can store field for each record.

	id	name	address	city	state	postal_code	telephone_number	email_address
	1	Byron de Villiers	5 Riepen Avenue, Riepen Park	Johannesburg	Gauteng	2061	0745815478	byron@mail.com
	2	Sheahan Hearn	11 High level Road	Cape Town	Western Cape	2001	0831237895	sheahan@mail.com
▶*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Tips for determining your columns:

- Don't include calculated data
- Store information in its smallest logical parts

Specifying primary keys

Each table should include a column or set of columns that uniquely identifies each row stored in the table. This is often a unique identification number, such as an employee ID number or a serial number. In database terminology, this information is called the **primary key** of the table.

	id	name	address	city	state	postal_code	telephone_number	email_address
	1	Byron de Villiers	5 Riepen Avenue, Riepen Park	Johannesburg	Gauteng	2061	0745815478	byron@mail.com
	2	Sheahan Hearn	11 High level Road	Cape Town	Western Cape	2001	0831237895	sheahan@mail.com
▶*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

In the example above the primary key could be the id and or email address and or telephone number.

Creating the table relationships

- **One-to-many**
 - To represent a one-to-many relationship in your database design, take the primary key on the "one" side of the relationship and add it as an additional column or columns to the table on the "many" side of the relationship.

Products:

	id	name	description	price	units_in_stock	supplier_id
▶	1	Chai tea	Drinking tea	50	500	1
	2	Rooibos tea	Drinking tea	100	200	1
	3	Earl gray tea	Drinking tea	150	150	1
*	NULL	NULL	NULL	NULL	NULL	NULL

Suppliers:

[illegible]

Creating the table relationships

- Many-to-many

Consider the relationship between the Products table and Orders table. A single order can include more than one product. On the other hand, a single product can appear on many orders. Therefore, for each record in the Orders table, there can be many records in the Products table. And for each record in the Products table, there can be many records in the Orders table. This type of relationship is called a many-to-many relationship.

Creating the table relationships

- Many-to-many

Orders:

	id	customer_id	total	date	paid
▶	1	1	500	12/03/2024	1
	2	1	600	12/03/2024	1
*	NULL	NULL	NULL	NULL	NULL

Products:

	id	name	description	price	units_in_stock	supplier_id
▶	1	Chai tea	Drinking tea	50	500	1
	2	Rooibos tea	Drinking tea	100	200	1
	3	Earl gray tea	Drinking tea	150	150	1
*	NULL	NULL	NULL	NULL	NULL	NULL

Order Products Mapping:

	id	order_id	product_id	qty
▶	1	1	1	5
	2	1	2	1
	3	1	3	1
	4	2	2	3
	5	2	3	2
*	NULL	NULL	NULL	NULL

Creating the table relationships

- One-to-one

Another type of relationship is the one-to-one relationship. For instance, suppose you need to record some special supplementary product information that you will need rarely or that only applies to a few products. Because you don't need the information often, and because storing the information in the Products table would result in empty space for every product to which it doesn't apply, you place it in a separate table. Like the Products table, you use the ProductID as the primary key. The relationship between this supplemental table and the Product table is a one-to-one relationship.

Creating the table relationships

- One-to-one

Products:

	id	name	description	price	units_in_stock	supplier_id
▶	1	Chai tea	Drinking tea	50	500	1
	2	Rooibos tea	Drinking tea	100	200	1
	3	Earl gray tea	Drinking tea	150	150	1
*	NULL	NULL	NULL	NULL	NULL	NULL

Product Info:

	id	product_id	volume	weight
▶	1	1	500ml	150
*	NULL	NULL	NULL	NULL

Refining the design

Once you have the tables, fields, and relationships you need, you should create and populate your tables with sample data and try working with the information: creating queries, adding new records, and so on. Doing this helps highlight potential problems — for example, you might need to add a column that you forgot to insert during your design phase, or you may have a table that you should split into two tables to remove duplication.

Refining the design

As you try out your initial database, you will probably discover room for improvement. Here are a few things to check for:

- Did you forget any columns?
- Are any columns unnecessary?
- Are you entering duplicate information?
- Has each information item been broken into its smallest useful parts?
- Does each column contain a fact about the table's subject?
- Are all relationships between tables represented?

UML

Unified Modeling Language.

UML - Introduction

Whats i **UML**?

- UML stands for Unified Modeling Language.
- It is a standardized visual language designed to depict the structure and behavior of software systems.
- UML provides a way to visualize a system's architectural blueprints, including elements like activities, actors, business processes, database schemas, and system components.

Types of UML diagrams

UML diagrams are categorized into two main groups: **Structural Diagrams**, and **Behavioral Diagrams**.

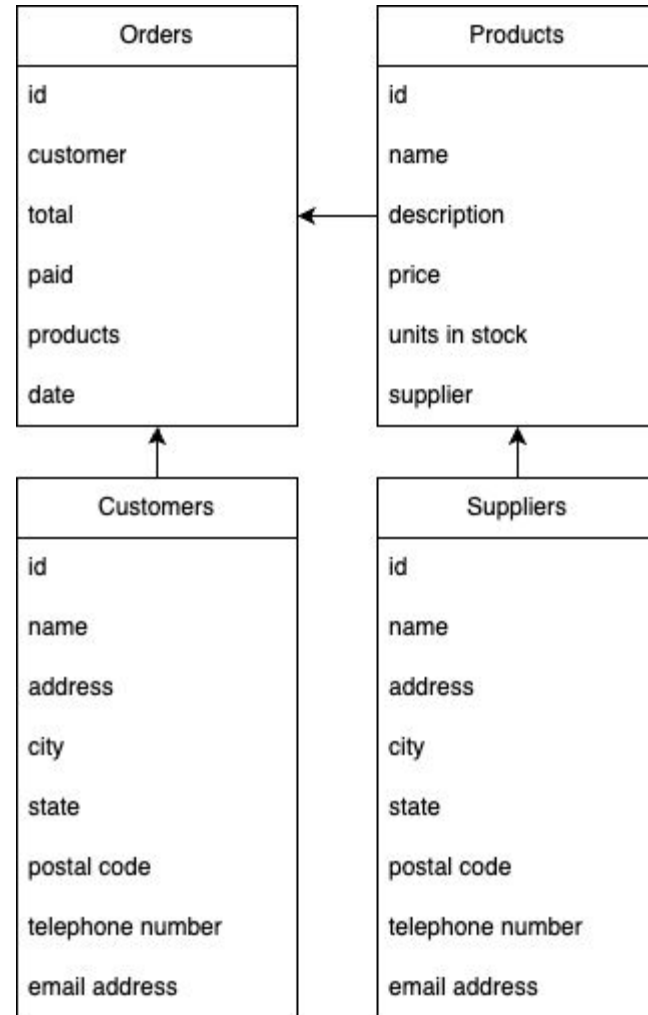
The most common UML diagrams are structural diagrams called **Class Diagrams**. It models the static structure of a system, showing classes, their attributes, operations (or methods), and the relationships among objects.

Class diagrams

Classes: Represented as boxes with three compartments for the class name, attributes, and methods.

Relationships: Including associations (simple connections), aggregations (whole/part relationships), compositions (stronger whole/part relationships with lifecycle dependency), and inheritances (generalization/specialization relationships).

- A Customer can place multiple Orders (one-many).
- Many Orders are related to many Products (many-many).
- One Supplier can supply many Products (one-many).



Exercise

Testing with Postman

Exercise:

- Create a blog model.
- Update each controller with blog model.
- Create a category model.
- Create a category model.
- Use Postman to test each api route:
 - Create
 - Read
 - Update
 - Delete
- Each response should align with the [tutorial results](#).
- Push changes to project Github repository.

Homework

Apply what we have learned

Homework

- Connect backend Express server with Mongo.
- Create a data model for blogs posts.
- Use Postman to add, read, update and delete data to local mongoDB server.

**Optional:*

- Updated front end blog app to fetch blog posts in the DB from the backend Express server.

Next Class

Database Design and Full Stack
Application

