



# Context

Code for today - <https://github.com/100xdevs-cohort-3/week-7-mongo>

In today's class, we'll understand about **databases**, and more specifically **NoSQL** databases.

We'll learn about MongoDB and how you can use it to persist data in your full stack app.

## Things to learn

1. Creating a free mongo db cloud server
2. Connecting your full stack application to MongoDB
3. Defining the schema
4. mongoose
5. CRUD operations





# What are databases

Databases are organized collections of data that are structured to enable efficient storage, retrieval, and management of information.

Whenever you create a **full stack app**, you **persist** data in **databases**.

For example –

1. User information
2. TODOs of your todo app
3. Posts for facebook
4. Tweets for twitter ...

## Types of databases

- **Relational Databases:** Use tables to store data and relationships between data (e.g., MySQL, PostgreSQL).

- **NoSQL Databases:** Designed for more flexible data models and often use MongoDB, Zod, Error handling and Input validation (e.g., MongoDB).
- **Graph Databases:** Store data in nodes and edges to represent relationships (e.g., Neo4j).
- **Vector Databases:** Used in ML to store data in the form of embeddings

Today we'll be learning about **MongoDB** which is a **NoSQL Database**

# MongoDB and NoSQL databases

NoSQL databases are a broad category of database systems that diverge from the traditional relational model used in SQL databases.

They are designed to handle a variety of data models and workloads that may not fit neatly into the tabular schema of relational databases.

## Properties

- **Schema Flexibility:** NoSQL databases often allow for a flexible schema, meaning you can store data in formats that don't require a fixed structure.
- **Scalability:** Many NoSQL databases are designed to scale out horizontally, making it easier to distribute data across multiple servers and handle large volumes of traffic.

# MongoDB



MongoDB, Zod, Error handling and Input validation 1 of 13

MongoDB is a NoSQL database that uses a document-oriented approach. Data is stored in flexible, JSON-like documents, which can have nested structures and varied fields.

## Creating a free MongoDB Server

1. Signup on ▶
2. Create a **free M0** cluster
3. Create a **User**
4. Install MongoDB compass
5. Put the connection URL to connect to the database



## Connection string

# Seeding data in the DB

Lets put some data in the **Cluster** manually

1. Create a new **Database** called **todo-app**
2. Create two collections inside it
  1. **users**
  2. **todos**
3. Seed some data inside the collections

## Users table



## TODO table

# CRUD Operations

CRUD operations in MongoDB refer to the basic operations you can perform on documents within a MongoDB database. CRUD stands for:

1. **Create:** Adding new documents to a collection.
2. **Read:** Retrieving documents from a collection.
3. **Update:** Modifying existing documents in a collection.
4. **Delete:** Removing documents from a collection.



# Creating the backend of a todo app

Lets now create a **todo application** with the data being **persisted** in the database.

- Initialise a new Node.js project

```
npm init -y
```



- Install dependencies

 ~~npm install express mongoose~~**MongoDB, Zod, Error handling and Input validation 1 of 13**

- Create the skeleton for 4 routes
  - POST /signup
  - POST /login
  - POST /todo (authenticated)
  - GET /todos (authenticated)

## ▼ Solution

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

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

app.post("/signup", function(req, res) {

});

app.post("/signin", function(req, res) {

});


app.post("/todo", function(req, res) {

});

app.get("/todos", function(req, res) {

});

app.listen(3000);
```



- Initialize the schema of your app in a new file (db.js)



## ▼ Easy schema



## MongoDB, Zod, Error handling and Input validation 1 of 13



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

const Schema = mongoose.Schema;
const ObjectId = Schema.ObjectId;

const User = new Schema({
  name: String,
  email: String,
  password: String
});

const Todo = new Schema({
  userId: ObjectId,
  title: String,
  done: Boolean
});

const UserModel = mongoose.model('users', User);
const TodoModel = mongoose.model('todos', Todo);

module.exports = {
  UserModel,
  TodoModel
}
```

## ▼ Hard schema

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

const Schema = mongoose.Schema;
const ObjectId = Schema.ObjectId;

const User = new Schema({
  name: String,
  email: {type: String, unique: true},
  password: String
});

const Todo = new Schema({
```





```
userId: ObjectId,  
title: String,  
done: Boolean  
});
```

```
const UserModel = mongoose.model('users', User);  
const TodoModel = mongoose.model('todos', Todo);  
  
module.exports = {  
  UserModel,  
  TodoModel  
}
```

- Import the model in `index.js`

```
const { UserModel, TodoModel } = require("./db");
```



- Implement the `/signup` endpoint

```
app.post("/signup", async function(req, res) {  
  const email = req.body.email;  
  const password = req.body.password;  
  const name = req.body.name;  
  
  await UserModel.create({  
    email: email,  
    password: password,  
    name: name  
  });  
  
  res.json({  
    message: "You are signed up"  
  })  
});
```



- Implement the `/signin` endpoint (need to install jsonwebtoken library)

```
const JWT_SECRET = "s3cret";
```



**MongoDB, Zod, Error handling and Input validation 1 of 13**

```
app.post("/signin", async function(req, res) {
```

```
  const email = req.body.email;  
  const password = req.body.password;
```

```
  const response = await UserModel.findOne({  
    email: email,  
    password: password,  
  });
```

```
  if (response) {  
    const token = jwt.sign({  
      id: response._id.toString()  
    })
```

```
    res.json({  
      token  
    })  
  } else {  
    res.status(403).json({  
      message: "Incorrect creds"  
    })  
  }  
});
```

- Implement the **auth** middleware (in a new file auth.js)

```
const jwt = require("jsonwebtoken");
```

```
const JWT_SECRET = "s3cret";
```

```
function auth(req, res, next) {  
  const token = req.headers.authorization;
```

```
  const response = jwt.verify(token, JWT_SECRET);
```

```
  if (response) {  
    req.userId = token.userId;  
    next();
```

```
  } else {  
    res.status(403).json({  
      message: "Incorrect creds"  
    })  
  }  
}
```





}}

MongoDB, Zod, Error handling and Input validation 1 of 13

```
module.exports = {  
  auth,  
  JWT_SECRET  
}
```

- Implement the **POST** todo endpoint

```
const { auth, JWT_SECRET } = require("./auth");
```



- Connect to your DB at the top of index.js

```
const mongoose = require("mongoose");  
mongoose.connect("mongodb+srv://gasfgfafa:Aa5jxKhylWdFhv4v@cluster0.7|
```



# Testing your app

Try testing your app in Postman next

## Signup endpoint



# Signin endpoint

# Create Todo

# Get todos



# Improvements

1. Password is not hashed
2. A single crash (duplicate email) crashes the whole app
3. Add more endpoints (mark todo as done)
4. Add timestamp at which todo was created/the time it needs to be done by
5. Relationships in Mongo
6. Add validations to ensure email and password are correct format



# Hashing password

## Why should you hash passwords?

Password hashing is a technique used to securely store passwords in a way that makes them difficult to recover or misuse. Instead of storing the actual password, you store a hashed version of it.

### salt

A popular approach to hashing passwords involves using a hashing algorithm that incorporates a salt—a random value added to the password before hashing. This prevents attackers from using precomputed tables (rainbow tables) to crack passwords.

### bcrypt

**Bcrypt:** It is a cryptographic hashing algorithm designed for securely hashing passwords. Developed by Niels Provos and David Mazières in 1999,

bcrypt incorporates a salt and is designed to be computationally expensive, making brute-force attacks more difficult.

MongoDB, Zod, Error handling and Input validation 1 of 13

## Base code

We're starting from yesterday's code - <https://github.com/100xdevs-cohort-3/week-7-mongo>

## Adding password encryption

- Install the `bcrypt` library - ▶
- Update the `/signup` endpoint

```
app.post("/signup", async function(req, res) {  
  const email = req.body.email;  
  const password = req.body.password;  
  const name = req.body.name;  
  
  const hashedPassword = await bcrypt.hash(password, 10);  
  
  await UserModel.create({  
    email: email,  
    password: hashedPassword,  
    name: name  
  });  
  
  res.json({  
    message: "You are signed up"  
  })  
});
```





## ▼ Password format



MongoDB, Zod, Error handling and Input validation 1 of 13

So, putting it all together:

- **\$2b\$** : Version of bcrypt.
  - **10\$** : Cost factor (saltRounds).
  - **wyemvgfpjkEzg2dzuRyM9e** : Salt value (base64 encoded).
  - **LrQZnT69X/tj0KW/zM6TZhnrvT.TCne** : Hashed password (base64 encoded).
- Update the **signin** function

```
app.post("/signin", async function(req, res) {  
  const email = req.body.email;  
  const password = req.body.password;  
  
  const user = await UserModel.findOne({  
    email: email,  
  });  
  
  const passwordMatch = bcrypt.compare(password, user.password);  
  if (user && passwordMatch) {  
    const token = jwt.sign({  
      id: user._id.toString()  
    }, process.env.JWT_SECRET, {  
      expiresIn: '1h'  
    });  
    res.json({ token });  
  }  
});
```



```
}, JWT_SECRET);
```



MongoDB, Zod, Error handling and Input validation 1 of 13

```
res.json({
  token
})
} else {
  res.status(403).json({
    message: "Incorrect creds"
  })
}
});
```

# Error handling

Right now, the server crashes if you sign up using duplicate email

How can you fix this?

## Approach #1 - Try catch

In JavaScript, a `try...catch` block is used for handling exceptions and errors that occur during the execution of code. It allows you to write code that can manage errors gracefully rather than crashing the application or causing unexpected behavior.

```
try {
  // Attempt to execute this code
  let result = riskyFunction(); // This function might throw an error
  console.log('Result:', result);
} catch (error) {
```



```
// Handle the error if one is thrown
```

```
    console.error('An error occurred:', error.message);  
  }  
  finally {  
    // This block will always execute  
    console.log('Cleanup code or final steps.');
```

```
  }
```

### ▼ Updated signin function

```
app.post("/signup", async function(req, res) {  
  try {  
    const email = req.body.email;  
    const password = req.body.password;  
    const name = req.body.name;  
  
    const hashedPassword = await bcrypt.hash(password, 10);  
  
    await UserModel.create({  
      email: email,  
      password: hashedPassword,  
      name: name  
    });  
  
    res.json({  
      message: "You are signed up"  
    })  
  } catch(e) {  
    res.status(500).json({  
      message: "Error while signing up"  
    })  
  }  
});
```



# Input validation

MongoDB, Zod, Error handling and Input validation 1 of 13

In TypeScript, Zod is a library used for schema validation and parsing. It's designed to help developers define, validate, and manage data structures in a type-safe manner.

Docs - <https://zod.dev/>

# Relationships in mongo

In MongoDB, data is **related** across collections using something called **references**

In our TODO application, each todo **refers** to an entry in the **users** table.

## Our original schema

```
const Todo = new Schema({
  userId: ObjectId,
  title: String,
  done: Boolean
});
```



## Update schema with references

```
const TodoSchema = new Schema({
  userId: { type: Schema.Types.ObjectId, ref: 'users' },
  title: String,
  done: Boolean
});
```



## Benefits?

You can **pre-populate** fields like **user information** since you've defined the exact relationship



## MongoDB, Zod, Error handling and Input validation 1 of 13

```
app.get('/todos', auth, async (req, res) => {  
  const userId = req.userId;  
  
  const todos = await TodoModel.find({  
    userId  
  }).populate('userId').exec();  
  
  res.json({  
    todos  
  })  
});
```