MERN 3 - Intro to MongoDB and Mongoose

Intro to MongoDB
Data storage in mongodb
Connecting to Mongodb atlas
Intro to Mongoose ODM : Model ,schema
CRUD with DB

title: Intro to Mongodb

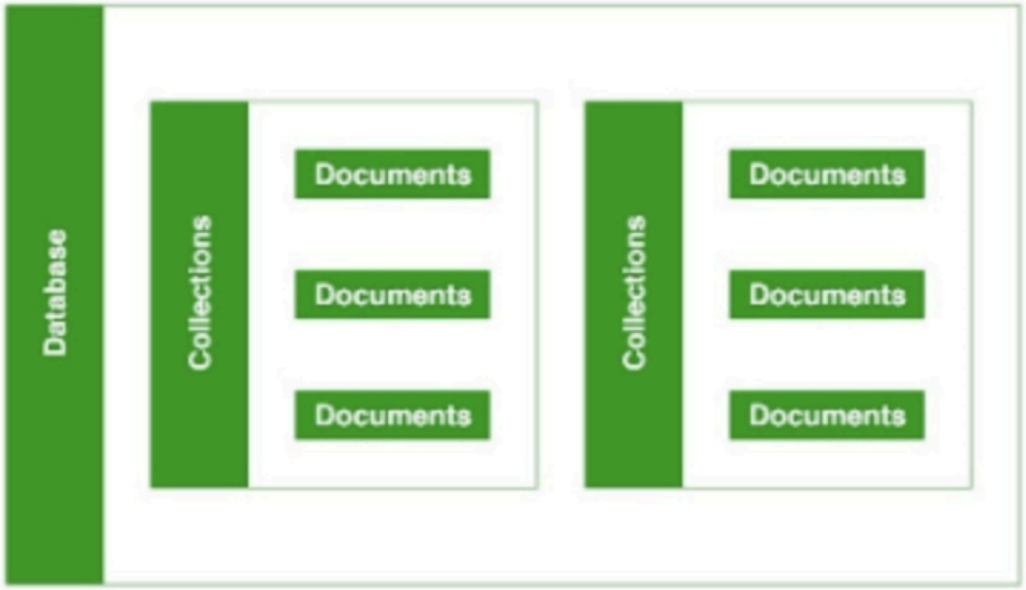
Database

1. A stupid question - Why do we need
    a. Store
    b. Retrieval
    c. Search, index
    d. Validation (some fields cannot be empty, structure of data being stored)
2. MOngoDB
    a. When you use MongoDB, you typically interact with the database using documents that look very much like JSON
    b. MongoDB stores data in a format called BSON, which stands for Binary JSON. BSON is a binary representation of JSON-like documents.
    c. This means the data is stored in a format that a computer can read very quickly. It's not in plain text like a normal

document or a JSON file, but in a special, compact format that computers can process more efficiently.

d. MongoDb is a type of NoSql database
    i. What is a NoSql database
        1. Imagine a library setting where everything is neatly kept at its place . everything has a specific place - SQL (structure is key)
        2. Now I want you to imagine my daughter's play area. There are toys neatly placed (by her mom), some are in groups like blocks or her dinner set and then there are some randomly lying around for quick access - NoSql
        3. To speak of it more formally, NoSQL databases are not primarily structured as a set of tables. They can store and manage data in formats such as key-value pairs, documents, graphs, or wide-column stores.
        4. Unlike SQL databases, which require a predefined schema and structured data, NoSQL databases allow the storage of data without a predefined schema. This makes them ideal for handling semi-structured or unstructured data.
e. MongoDB, an open-source document-oriented database, is purpose-built for efficiently handling extensive datasets. It falls within the NoSQL (Not only SQL) database category

due to its departure from the conventional table-based storage and retrieval of data.



f.

g. Data in MongoDB is organized into collections and documents. This establishes a hierarchical relationship between databases, collections, and documents.
h. Collections are like containers or folders in which MongoDB stores related documents. You can think of them as similar to tables in a traditional SQL database, but with a more flexible structure.
i. Documents are the basic unit of data in MongoDB. They're similar to rows or records in a table, but unlike SQL databases where each row follows a fixed schema, MongoDB documents can have varying structures within the same collection.

EXTRA: Types of NoSql DB

1. Types of NoSQL Databases:
    a. Document Databases: Store data in documents similar to JSON objects. Each document contains pairs of fields and values. Examples include MongoDB and CouchDB.
    b. Key-Value Stores: Store data as a collection of key-value pairs. Examples include Redis and DynamoDB.
    c. Wide-Column Stores: Store data in tables, rows, and dynamic columns. They are excellent for querying large datasets.
        i. Examples include Apache Cassandra and Google Bigtable.
        ii. Imagine a wide-column store as a very flexible and large table. Just like in a regular table, you have rows and columns, but with some special features:
        iii. Tables, Rows, and Columns: Think of a big spreadsheet. You have rows for each item (like a person or a product), and columns for the details about them (like name, price, color).
        iv. Dynamic Columns: Unlike a regular table where each row must have the same columns, in a wide-column store, each row can have its own set of columns. It's like if each person in your contact list could have different kinds of information. One friend might have an address and phone number, while another has an

email and birthday, and another one has completely different information.

v. Great for Large Datasets: Because of this flexibility and the way it's structured, it's really good for handling lots of data. This is especially true when you have lots of different kinds of data that don't fit neatly into the same format for each item.

vi. In summary, wide-column stores are like super-flexible spreadsheets where each row can have its own set of unique columns, making them great for managing and querying large and diverse sets of data.

vii.

d. Graph Databases:

i. Neo4j

Example

Real-World Example:
Let's consider a scenario where you're building an e-commerce platform. You have a MongoDB database to store product information.

Collection: Products
This collection stores information about various products available on your platform.

Documents: Product Information

Each document represents a specific product and can have different fields based on the product type. For instance:

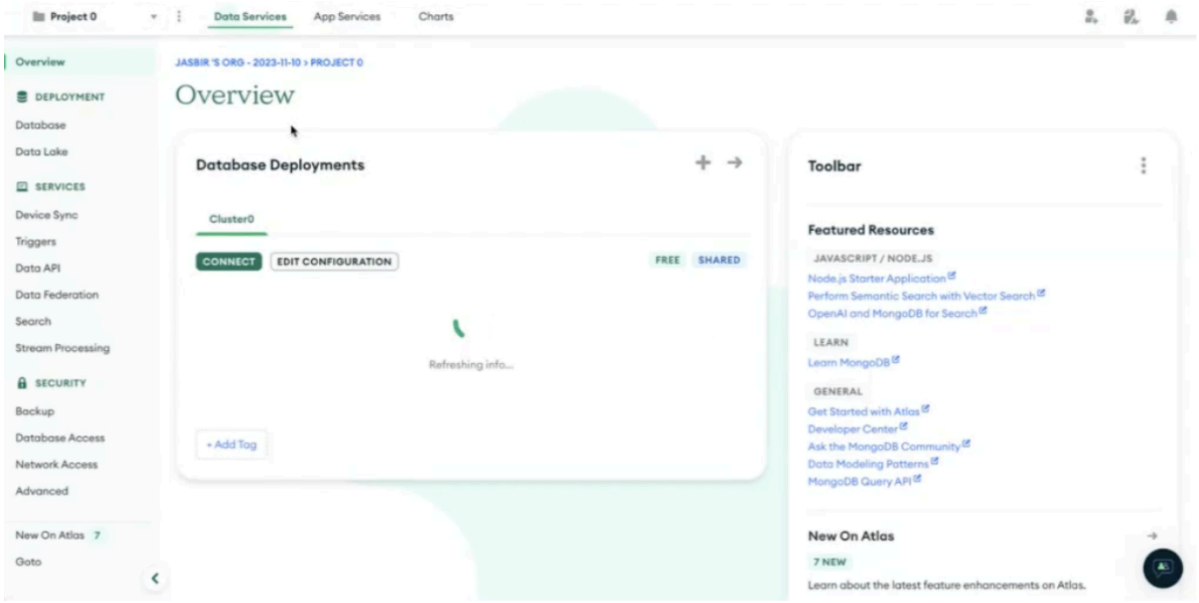
```
{
  "_id": ObjectId("61e65529b6fc4670e05a1c7a"),
  "name": "Smartphone",
  "brand": "XYZ",
  "price": 699,
  "specs": {
    "display": "6.5 inches",
    "storage": "128GB",
    "camera": "Quad-camera setup"
  }
}
```

Here, the Products collection holds documents representing different types of products. This document represents a smartphone with its name, brand, price, and specifications.

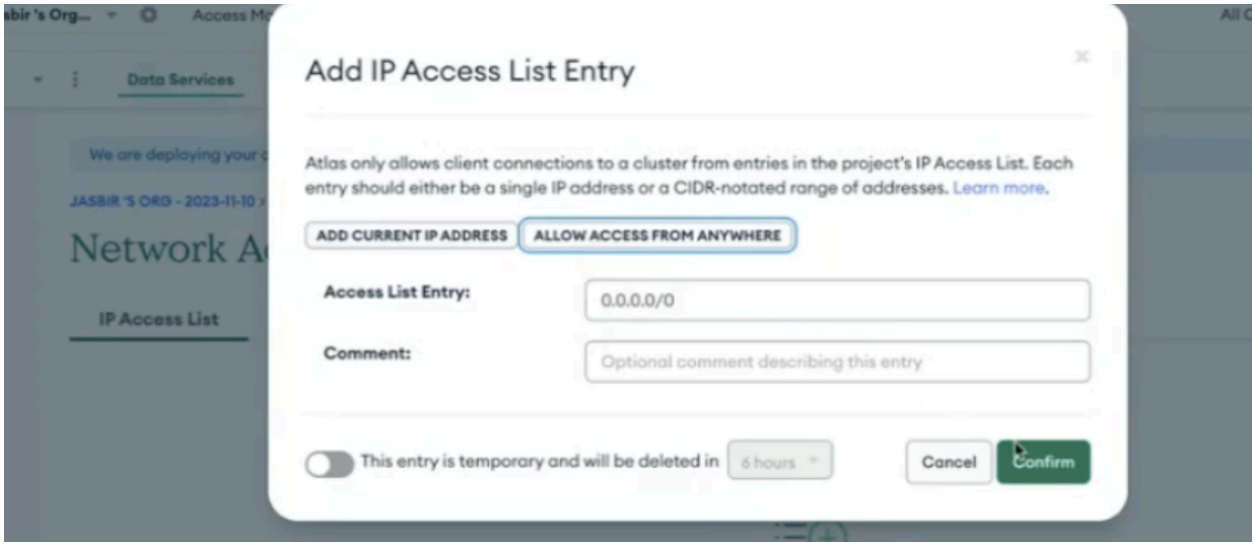
```
{
  "_id": ObjectId("61e65545b6fc4670e05a1c7b"),
  "name": "Laptop",
  "brand": "ABC",
  "price": 1299,
  "specs": {
    "display": "15.6 inches",
```

```
    "storage": "512GB SSD",
    "RAM": "16GB",
    "processor": "Intel Core i7"
  }
}
```

Steps to get DB Link from Mongodb Atlas

1. Signup to mongodb.com
2. follow the steps and create a cluster
3. once cluster is created you will see this dashboard screen
4. 

5. Usually server connecting with DB server has satatic IP and for security reasons this network access should only be given to creatin Ip address but for develop enviornment we can allow it to acces from anywhere. In network access select allow access from anywhere

6. 

7. Create a DB user , create it's password
8. Select Role as atlas admin
9. Go to Home Page and select connect . Choose Driver for node and copy the link

title: connecting MongoDB database with Express server

1. We use Mongoose to connect to our DB
2. "Mongoose is a tool for Node.js that helps you work with MongoDB. It makes some of the more complicated parts of using MongoDB easier.
3. Mongoose is like an assistant that helps you manage and use your data in MongoDB, making sure everything is organized and correct."
4. Mongoose acts as an intermediary between your Express server and MongoDB. It helps define data models, schemas, and provides methods for interacting with MongoDB. It simplifies the

process of querying and manipulating data in MongoDB from your Express application.

Connect to DB

```
const mongoose = require('mongoose');
const express = require("express");
const app = express();
// driver

const dbURL =
`mongodb+srv://ayushrajsd:28Wca5DmXC6Q9BDW@cluster0.vg5saeo.mongodb.net/`;
// once
mongoose.connect(dbURL)
   .then(function (connection) {
       console.log("connected to db",connection)
   }).catch(err => console.log(err))

 app.listen(PORT, () => {
   console.log(`The server is running in port ${PORT}`);
 });
```

title: creating schema and model

MongoDB Entity, Schema and Models

1. Entity
    a. An entity is like an object or a thing in the real world that has information we want to store. For example, if we are making a database for a school, student data can be an entity.

    b. For ecommerce website, products can be an entity, users, reviews can be different entities
    c. Different category of data
2. Schema
    a. A schema in MongoDB defines the structure of the data for an entity.
    b. Consider a schema as a form that a student fills out on their first day at school.
    c. In addition, It has certain rules and requirements that has to be followed like certain fields are required, data type constraints , etc
    d. To make it more clear user as a schema for Netflix will be very different than from say Amazon.
3. Model
    a. A model in MongoDB is a high-level programming interface that lets you interact with the data corresponding to a schema. It's like a tool to create, read, update, and delete entities in the database.
    b. Think of Mongoose models like cookie cutters. Just as a cookie cutter shapes dough into a specific form, a Mongoose model shapes your data to fit a certain structure before it's saved into a database.
    c. It is an instance of a schema and represents a collection of documents in the database that adhere to the schema's structure.

d. The model serves as the interface for interacting with the database, allowing you to perform CRUD (Create, Read, Update, Delete) operations and query the database.

Defining a Schema and Model

```
// schema
const productSchema = new mongoose.Schema({
   product_name: {
     type: String,
     required: true,
   },
   product_price: {
     type: String,
     required: true,
   },
   isInStock: {
     type: Boolean,
     required: true,
   },
   category: {
     type: String,
     required: true,
   },
 }, { timestamps: true });
  const ProductModel = mongoose.model("products", productSchema);
```

Explanation:

mongoose.Schema:

This is a constructor provided by Mongoose to define a new schema. It takes an object that maps the structure of the documents in a collection.

product_name:
type: String: Specifies that the product_name field should be of type String.
required: true: This field is mandatory, and Mongoose will throw a validation error if this field is not provided when creating a new document.

product_price:
type: String: Specifies that the product_price field should be of type String. (Typically, you'd use Number for prices, but this example uses String.)
required: true: This field is also mandatory.

isInStock:
type: Boolean: Specifies that the isInStock field should be of type Boolean.
required: true: This field is mandatory.

category:
type: String: Specifies that the category field should be of type String.
required: true: This field is mandatory.

Schema Options:
timestamps: true: This option automatically adds two fields to the schema: createdAt and updatedAt. These fields store the timestamps for when the document was created and last updated, respectively.

Model: A model is a class that we construct from a schema. It provides an interface to interact with the database and perform CRUD operations.

By calling mongoose.model("products", productSchema), we create a model named ProductModel associated with the products collection in the database. Mongoose automatically pluralizes the model name to determine the collection name.

Now, ProductModel can be used to interact with the products collection:

Whole code so far

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

// driver

const dbURL =
`mongodb+srv://ayushrajsd:28Wca5DmXC6Q9BDW@cluster0.vg5saeo.mongodb.net/`;
// once

mongoose.connect(dbURL)
   .then(function (connection) {
       console.log("connected to db",connection)
```

```
  }).catch(err => console.log(err))

// schema
const productSchema = new mongoose.Schema({
  product_name: {
    type: String,
    required: true,
  },
  product_price: {
    type: String,
    required: true,
  },
  isInStock: {
    type: Boolean,
    required: true,
  },
  category: {
    type: String,
    required: true,
  },
}, { timestamps: true });
 const ProductModel = mongoose.model("products", productSchema);

app.use(express.json()); // middleware for post request

app.listen(PORT, () => {
  console.log(`The server is running in port ${PORT}`);
});
```

So now DB connection , schema and model is ready

Now Let's see how to do CRUD with Mongoose

title: Database CRUD

Creating a Product

```
app.post("/api/products", async (req, res) => {
  const body = req.body
```

```
    const product = await ProductModel.create({
      product_name : body.product_name,
      product_price : body.product_price,
      isInStock : body.isInStock,
      category : body.category
    })
    console.log(product)
    return res.status(201).json({message : 'Product Created'})

});
```

Get Products and get a Single Product

```
app.get("/api/products", async (req, res) => {
 const allProducts = await ProductModel.find({})
 // const allProducts = await ProductModel.find({isInStock :true})
 console.log(allProducts)

 return res.status(200).json(allProducts);
});
```

Create another product in a different category

And then add this filter to see only electronics products

```
app.get("/api/products", async (req, res) => {
 const allProducts = await ProductModel.find({category : "electronics"})
 // const allProducts = await ProductModel.find({isInStock :true})
 console.log(allProducts)

 return res.status(200).json(allProducts);
});
```

Finding a Product by ID:

```
app.get("/api/products/:id", async (req, res) => {
 const id = req.params.id;
 const product = await ProductModel.findById(id);
 res.status(200).json(product);
```

```
})
```

Query from POSTMAN - `localhost:3000/api/products/666573f5ac33e20c0f721bc7`

Update a product in the DB (put method)

```
app.put("/api/products/:id", async (req, res) => {
 await ProductModel.findByIdAndUpdate(req.params.id , req.body)
 return res.status(201).json({message : 'Resources Updated'})
});
```

Delete a product in the DB

```
app.delete("/api/products/:id", async (req, res) => {
 await ProductModel.findByIdAndDelete(req.params.id)
 return res.status(201).json({message : 'Resource Deleted'})
});
```

Validator function

Sometimes we need to add small logic for our fields like password and confirmPassword should match. Or discount should be less than price etc. Such small logic can be handled in the schema before it is saved in DB

```
password: {
    type: String,
    required: true,
    minLength: 8,
  },
  confirmPassword: {
    type: String,
```

```
    required: true,
    minLength: 8,
    validate: {
      validator: function () {
        return this.password === this.confirmPassword;
      },
      message: "Password and confirm password should be same",
    }
  },
```

Make changes in the POST route

```
const product = await ProductModel.create({
      product_name : body.product_name,
      product_price : body.product_price,
      isInStock : body.isInStock,
      category : body.category,
      password : body.password,
       confirmPassword : body.confirmPassword
    })
```

Test in POSTMAN

Also discuss wrapping the code in try catch blocks

```
app.post("/api/products", async (req, res) => {
  const body = req.body
  try{

    const product = await ProductModel.create({
      product_name : body.product_name,
      product_price : body.product_price,
      isInStock : body.isInStock,
      category : body.category,
      password : body.password,
       confirmPassword : body.confirmPassword
    })
     console.log(product)
     return res.status(201).json({message : 'Product Created'})
  }catch(err){
```

```
    console.log(err)
    return res.status(400).json({message : 'Something went wrong',err})
  }

});
```

If you see we have a populated as well as we have polluted the whole code , DB connection is in the same file , models and schema are in the same file , all the CRUD operations are in the same file

As our project goes larger this kind of code can become very hard to mainatain

So to maintain a cleaner approach we will discuss MVC architecture in the Next class and will make everything organized