Database Vs Schema

- A database is the overall container for data, while a schema is a logical grouping within that database that defines the structure and organization of the data.
- Multiple schemas can exist within a single database, allowing for better organization and management of related data objects.
- In the context of databases, "database" and "schema" are related but distinct concepts.

Database

- A database is a structured collection of data that is organized for efficient retrieval, storage, and manipulation.
- It's a container that holds multiple tables, views, procedures, functions, and other database objects.
- Databases are typically managed by Database Management Systems (DBMS) like MySQL, PostgreSQL, Oracle, SQL Server, etc.
- Examples of databases include "company_database," "school_database," "inventory_system," etc.

Schema

- A schema is a logical container within a database that defines the structure, organization, and constraints of the data stored in the database.
- It defines the blueprint for how data is organized into tables, or documents the relationships between tables or documents, data types, constraints, indexes, etc.
- A schema can be thought of as a namespace within a database, providing a way to group related objects together.
- In some database systems, like PostgreSQL or MySQL, a schema is equivalent to a namespace, and multiple schemas can exist within a single database.

For example, within the "company_database," you might have schemas like "employee_data," "sales_data," and "customer_data," each containing tables and other objects relevant to their respective domains.

Mongoose

- Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js.
- It provides a higher-level abstraction over the MongoDB Node.js driver, making it easier to work with MongoDB databases by defining schemas for your data models and providing methods for interacting with those models.
- It simplifies the process of working with MongoDB databases in Node.js applications by providing a rich set of features for data modeling, validation, querying, and middleware execution.
- It is widely used in the Node.js ecosystem for building scalable and maintainable applications with MongoDB as the database backend.

Key features and functionalities of Mongoose

- **Schema-based modeling:** Mongoose allows you to define schemas for your data models. A schema defines the structure of documents within a collection, including the fields and their types, default values, validation rules, and more. This helps maintain consistency and structure in your data.
- **Data validation:** Mongoose provides built-in support for data validation, allowing you to define validation rules for each field in your schema. You can specify data types, required fields, custom validators, and more, ensuring that data entered into the database meets your application's requirements.
- **Middleware:** Mongoose supports middleware functions that allow you to execute custom logic before or after certain operations, such as saving documents, updating documents, and removing documents. Middleware can be useful for tasks like encryption, data manipulation, and logging.
- **Query building:** Mongoose provides a powerful query builder API that simplifies the process of querying MongoDB databases. It offers methods for performing CRUD (Create, Read, Update, Delete) operations, as well as advanced querying options such as filtering, sorting, limiting, and pagination.
- **Population:** Mongoose supports population, which allows you to reference documents from other collections and automatically populate them when querying. This helps in handling relationships between data models and avoids the need for manual data retrieval.
- **Integration with Express.js:** Mongoose is often used in conjunction with the Express.js web framework for building Node.js applications. It seamlessly integrates with Express.js, allowing you to define data models, perform database operations, and handle data validation within your Express.js routes and controllers.

Integrate mongoose with NodeJS and MongoDB

Integrating Mongoose with a Node.js application and MongoDB involves several steps, including setting up the project, defining schemas, connecting to the MongoDB database, performing CRUD operations, and handling errors.

You need to follow the below steps to integrate Mongoose with a Node.js application:

- **Initialize Node.js project**
  First, create a new directory for your project and initialize a Node.js project by running the following commands in your terminal:
  ```
  mkdir myapp
  cd myapp
  npm init -y
  ```

- **Install dependencies**
  Install Mongoose, which is the primary dependency for working with MongoDB in Node.js applications:
  ```
  npm install mongoose
  ```

- **Create a MongoDB database**
  Make sure you have MongoDB installed and running on your system. If not, you can download and install MongoDB from the official website
  https://www.mongodb.com/try/download/community

- **Define a schema**
  Define a Mongoose schema to model your data. Create a new file, user_model.js, and define a user schema:
  ```
  const mongoose = require('mongoose');

  const userSchema = new mongoose.Schema({
    username: { type: String, required: true },
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true }
  });

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

- **Connect to MongoDB**
  In your main application file (e.g., app.js), connect to your MongoDB database using Mongoose:

```
const mongoose = require('mongoose');
// Connect to MongoDB
mongoose.connect('mongodb://localhost:27016/myapp',
          {
            serverSelectionTimeoutMS: 5000 // Set a timeout of 5
seconds
          })
.then(() => {
  console.log('Connected to MongoDB');
})
.catch((error) => {
  console.error('Error connecting to MongoDB:', error);
});
```

  Replace 'mongodb://localhost:27017/myapp' with the connection string to your MongoDB database.

- **Perform CRUD operations**
  Now you can perform CRUD operations using Mongoose methods.
  For example, to create a new user:

```
const User = require('./user_model');
// Create a new user
const newUser = new User({
  username: 'shruti khandelwal',
  email: 'shrutikh129@gmail.com',
  password: 'test@123'
});
newUser.save()
  .then((user) => {
    console.log('User created:', user);
  })
  .catch((error) => {
    console.error('Error creating user:', error);
  });
```

For example, to find an existing user:

```
// Find a user based on email ID
User.findOne({ email: 'shrutikh129@gmail.com'}).exec()
.then(user => {
    if (user) {
        console.log('User found:', user);
    } else {
        console.log('User not found');
    }
})
.catch(err => {
    console.error('Error finding user:', err);
});
```

For example, to update an existing user:

```
// Find the user you want to update and update their email
    User.findOneAndUpdate(
        { email: 'shrutikh129@gmail.com'}, // Query to find the user
        { email: 'shruti@test.com' }, // New email value to set
        { new: true } // To return the modified document rather than the
original
    )
    .then(updatedUser => {
        if (updatedUser) {
            console.log('User updated:', updatedUser);
        } else {
            console.log('User not found');
        }
    })
    .catch(error => {
        console.error('Error updating user:', error);
    });
```

For example, to delete an existing user:

```
// Delete the user with the specified email
  User.deleteOne({ email: 'shruti@test.com' })
  .then(result => {
    if (result.deletedCount > 0) {
      console.log('User deleted successfully');
    } else {
      console.log('User not found');
    }
  })
  .catch(error => {
    console.error('Error deleting user:', error);
  });
```

"Now you have a little bit idea about how to integrate Mongoose with MongoDB and Node.js, and how to perform CRUD operations."

Schemas Type

- There are the two main types of schemas used in Mongoose to define the structure of documents in MongoDB collections.
- Regular schemas define the top-level structure, while nested schemas allow for more complex document structures with subdocuments.

Regular Schemas

- Regular schemas are the primary way to define the structure of documents (data) in MongoDB collections using Mongoose.
- They define the shape of documents, including the fields and their types, default values, validation rules, and other options.
- Regular schemas are defined using the new mongoose.Schema() constructor function and then compiled into a model using mongoose.model().

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

// Define a regular schema
const userSchema = new mongoose.Schema({
    name: String,
    age: Number,
    email: {
                type: String,
                required: true,
                 unique: true }
});

// Compile the schema into a model
const User = mongoose.model('User', userSchema);
```

Nested Schemas

- Nested schemas allow you to define subdocuments within a document's structure.
- They are useful for organizing data hierarchically and encapsulating related fields.
- Nested schemas are defined similarly to regular schemas but are embedded within another schema definition.

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

// Define a nested schema {set}for address
const addressSchema = new mongoose.Schema({
   street: String,
   city: String,
   state: String,
   country: String
});

// Define a regular schema containing the nested schema
const userSchema = new mongoose.Schema({
   name: String,
   age: Number,
   email: {
              type: String,
              required: true,
              unique: true },
   address: addressSchema  // Nested schema as a field
});

// Compile the schema into a model
const User = mongoose.model('User', userSchema);
```

CRUD Operations with Node.js, MongoDB, and Mongoose

Objective: To build an **E-Commerce Backend** directly in a single file (app.js), using **Node.js**, **MongoDB**, and **Mongoose**, with a mix of **regular schemas** and **nested schemas**. The goal is to gain advanced practical experience in CRUD operations, validation, and complex relationships without using routes or MVC architecture.

Requirements

1. Models

- **User Model**

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, enum: ["Admin", "Customer"], default: "Customer" },
  createdAt: { type: Date, default: Date.now }
});
const User = mongoose.model('User', userSchema);
```

- **Product Model**

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  description: { type: String, required: true },
  price: { type: Number, required: true, min: 0 },
  category: { type: String, required: true },
  stock: { type: Number, required: true, min: 0 },
  createdAt: { type: Date, default: Date.now }
});
const Product = mongoose.model('Product', productSchema);
```

- **Order Model** (Nested Schema)

```
const orderSchema = new mongoose.Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true },
  products: [
    {
      product: { type: mongoose.Schema.Types.ObjectId, ref: "Product",
required: true },
      quantity: { type: Number, required: true, min: 1 }
    }
  ],
  totalAmount: { type: Number, required: true },
  orderDate: { type: Date, default: Date.now },
  status: { type: String, enum: ["Pending", "Shipped", "Delivered",
"Cancelled"], default: "Pending" }
});
const Order = mongoose.model('Order', orderSchema);
```

2. Assignment Tasks

- **Setup**
  1. Create a single app.js file.
  2. Use mongoose.connect() to connect to MongoDB.
  3. Use express.json() for request parsing.
- **User CRUD Operations**
  1. Create a user with name, email, password, and role.
  2. Fetch all users.
  3. Fetch a specific user by their email or id.
  4. Update a user's name or role by id.
  5. Delete a user by id.
- **Product CRUD Operations**
  1. Create a product with name, description, price, category, and stock.
  2. Fetch all products with filters for category and price range (e.g., min and max price).
  3. Fetch a specific product by id.
  4. Update a product's price or stock by id.
  5. Delete a product by id.
- **Order CRUD Operations**
  1. Create an order for a user:
     1. Validate that all product IDs exist and stock is sufficient.
     2. Decrease the stock of the ordered products.
     3. Calculate totalAmount as quantity * price for each item.
  2. Fetch all orders, including user and product details.
  3. Fetch a specific order by its id, with populated user and product details.

4. Update an order's status (e.g., "Shipped", "Delivered").
5. Delete an order and restore the stock for its products.

Validation Requirements

1. Ensure
    - Unique email for the User model.
    - Positive values for price, stock, and quantity.
2. Validate totalAmount dynamically in the Order model.
3. Prevent deletion of a product that is referenced in an active order.

Date of Submission : 24 Jan 2025 before 13:59:59