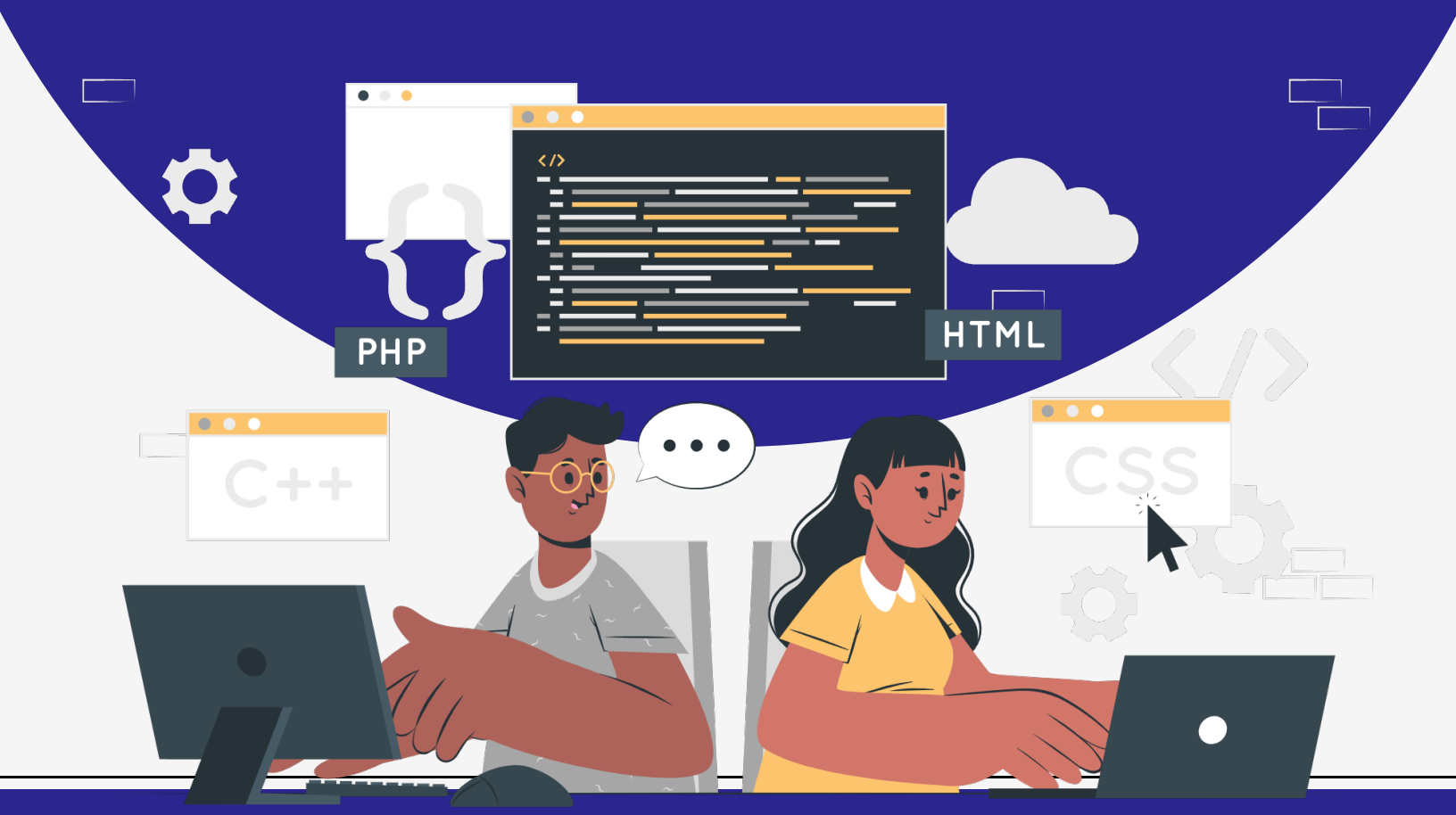
Lesson:

Mongoose ODM

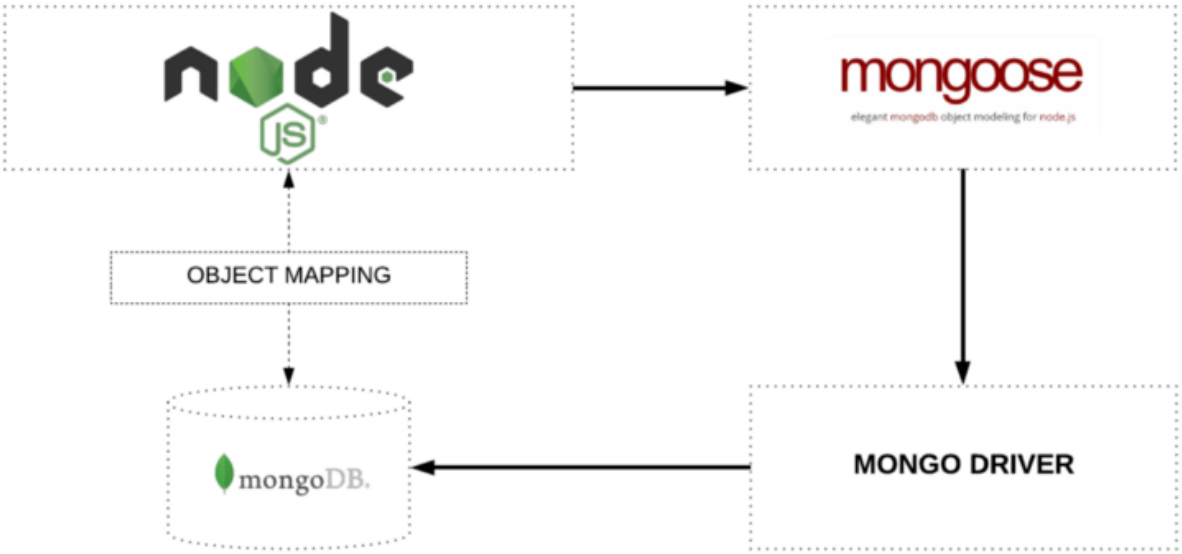


PW SKILLS

Topics

- What is Mongoose
- Connecting to the MongoDB database
- Model and Schema with Mongoose
- Data insertion with Mongoose
- Mongoose constraints or Validation
- Data fetching with Mongoose
- Update Data with Mongoose
- Delete Data with Mongoose
- Relationship
  - Embedding or Nesting collections
  - Referencing

What is Mongoose



Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a higher-level abstraction for interacting with MongoDB databases, allowing developers to define schemas, models, and relationships between data in a structured and organized manner. Mongoose simplifies common tasks such as connecting to a MongoDB database, defining schemas to enforce data integrity, and performing CRUD operations.

For example, suppose you're building a blogging platform using Node.js and MongoDB. With Mongoose, you can define schemas for blog posts, comments, and users.
Here's an example of how you might define a schema for a blog post using Mongoose:

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

// Define a schema for blog posts
const postSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true }});

// Create a model based on the schema
const Post = mongoose.model('Post', postSchema);

module.exports = Post;
```

In the above example, we define a schema for users with username, email, and password fields. We then create a model named 'User' based on this schema, which can be used to interact with the 'users' collection in the MongoDB database, it might seem a bit complex at first glance. However, don't worry, we'll delve into each topic in more detail in the following topics. We'll break down the concepts and explain them thoroughly, making it easier for you to understand and apply them in your projects. Let's explore each topic step by step to ensure clarity and comprehension.

Connecting to the MongoDB database

Connecting to a MongoDB database using Mongoose is a crucial step in building Node.js applications. You'll need to provide the connection URL and handle connection events to establish a connection.

Before we initiate the connection to the MongoDB database, it's essential to ensure that the Node.js project is configured with Mongoose installed, along with Nodemon for automatic project restarting.

Create Node.js project
Run the command below to create the Node.js project

```
npm init -y
```

Install Mongoose and Nodemon

```
# npm i monoose and nodemon
```

Establish MongoDB database connection.
Here's how you can connect to a MongoDB database using Mongoose
index.js

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

// Connection URL
const MONGODB_URI = 'mongodb://127.0.0.1:27017/db';

// Connect to MongoDB
mongoose.connect(MONGODB_URI, {});

// Handle connection events
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'Connection error:'));
db.once('open', () => {
  console.log('Connected to MongoDB');
});
```

Run the Node.js project
To run the node.js project, modify the node package.json with nodemon as given below -

```
{
  "name": "mongoose",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mongoose": "^8.2.1",
    "nodemon": "^3.1.0"
  }
}
```

Now run the command given below to start the application

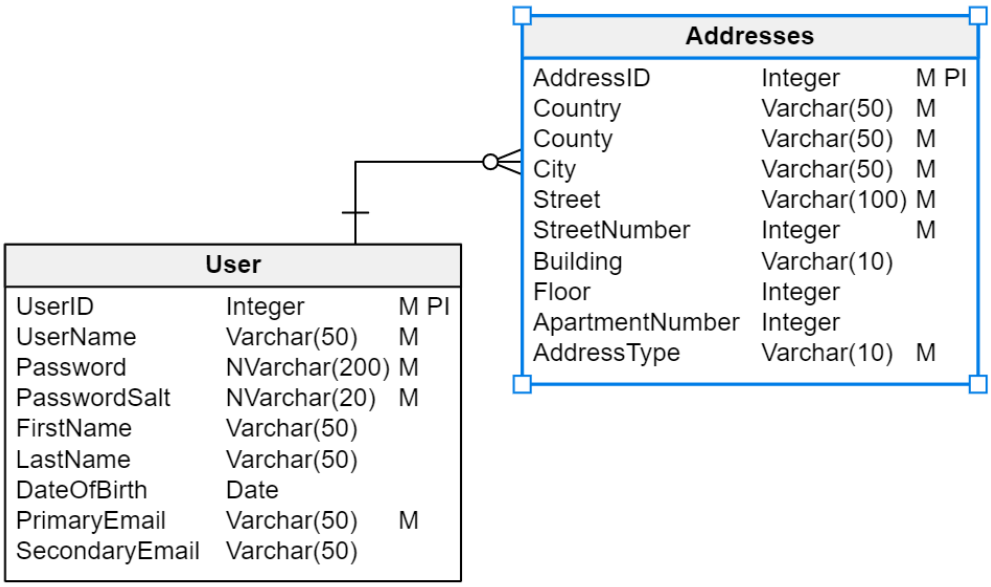
```
npm run dev
```

If it runs successfully, you'll see the output as given below

```
Ineuron@LAPTOP-K39S8VLG MINGW64 /d/Mang/Learnings/backend/mongoose (main)
$ npm run dev


[nodemon] 3.1.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Connected to MongoDB
```

From the above example, we provide the connection URL ('mongodb://127.0.0.1:27017/db') and use the mongoose.connect() method to establish a connection to the MongoDB database. We handle connection errors and log a success message once the connection is open.

Model and Schema with Mongoose



In Mongoose, models and schemas play a fundamental role in defining the structure and behavior of data stored in MongoDB collections.

For example, Let's define a schema and model for users:
users.model.js

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

// Define a schema for users
const userSchema = new mongoose.Schema({
  username: String,
  email: String,
  password: String,
});

// Create a model based on the schema
const User = mongoose.model('User', userSchema);

module.exports = User;
```

From the above example, we define a schema for users with username, email, and password fields. We then create a model named **'User'** based on this schema, which can be used to interact with the **'users'** collection in the MongoDB database.

Data insertion with Mongoose

Inserting data into MongoDB collections using Mongoose involves creating instances of models and saving them to the database.

For example, Let's insert a new user into the **"users"** collection
index.js

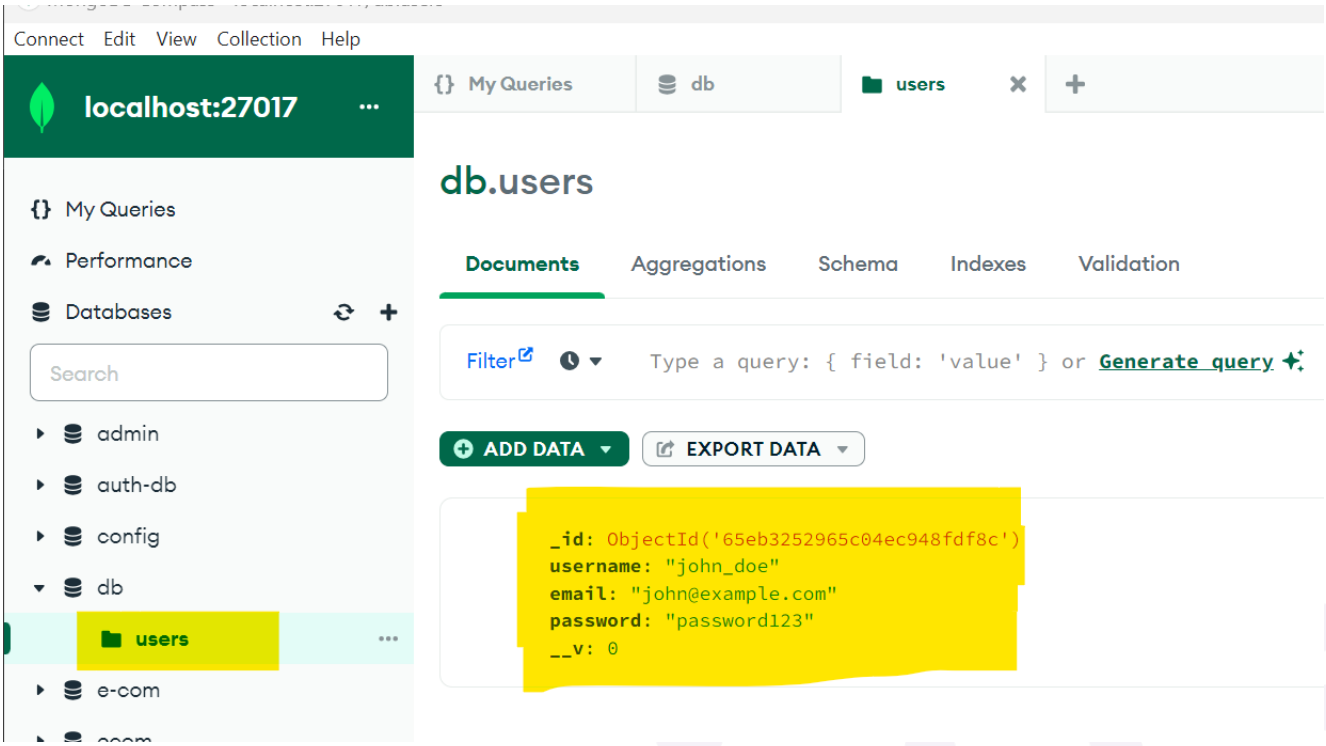
```
// create or insert user
const createUser = async () => {
  try {
    // Create a new user instance
    const newUser = new User({
      username: 'john_doe',
      email: 'john@example.com',
      password: 'password123',
    });

    // Save the user to the database
    const user = await newUser.save();
    console.log('User saved successfully:', user);
  } catch (error) {
    console.error(`User failed to save: ${error}`);
  }
};
```

If the above code successfully run, you will get the output as shown below

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Connected to MongoDB
[nodemon] restarting due to changes...
Connected to MongoDB
User saved successfully: {
  username: 'john_doe',
  email: 'john@example.com',
  password: 'password123',
  _id: new ObjectId('65eb3252965c04ec948fdf8c'),
  __v: 0
}
```

You can also confirm that the user has been inserted successfully with MongoDB compass or Mongosh as well

In MongoDB compass -



In mongoose terminal -

```
test> show dbs
admin        40.00 KiB
auth-db     108.00 KiB
config      108.00 KiB
db           72.00 KiB
e-com       112.00 KiB
ecom          8.00 KiB
local        80.00 KiB
my-db        72.00 KiB
mydatabase   40.00 KiB
profilesdb   60.00 KiB
test> use db
switched to db db
db> show collections
users
db> db.users.find()
[
  {
    _id: ObjectId("65eb3252965c04ec948fdf8c"),
    username: 'john_doe',
    email: 'john@example.com',
    password: 'password123',
    __v: 0
  },
```

Mongoose constraints or Validation

In Mongoose, we can add constraints to enforce data integrity and ensure that the data stored in the MongoDB collection meets certain criteria.
Let's discuss the constraints we can add to each field:

Username constraints
We can enforce constraints on the username field to ensure that it is required and unique for each user. This prevents users from creating accounts without specifying a username and ensures that usernames are unique within the database.
i.e

```
username: { type: String, required: true, unique: true }
```

Email constraints
For the email field, we can enforce constraints to ensure that it is required, unique, and follows a valid email format. This prevents users from registering without providing an email address, ensures that each email address is unique, and validates that the email format is correct.
i.e

```
email: { type: String, required: true, unique: true}
```

Password Constraints
For the password field, we typically want to ensure that it is required and meets certain complexity requirements, such as a minimum length or the presence of special characters. This helps enhance security by requiring users to choose strong passwords.

```
password: { type: String, required: true, minlength: 6 }
```

Now the User Schema will look like as given below -
users.model.js

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

// Define a schema for users
const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true, minlength: 6 },
});

// Create a model based on the schema
const User = mongoose.model('User', userSchema);

module.exports = User;
```

With these constraints added to the user schema, the User model will enforce these rules when creating or updating user documents. Any attempt to save a document that violates these constraints will result in a validation error, ensuring that only valid data is stored in the MongoDB collection.
For example, In the above schema username field is required and you will get the error message as shown below in case you failed to provide the username field.

```
[nodemon] starting `node index.js`
Connected to MongoDB
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Connected to MongoDB
User failed to save: ValidationError: username: Path `username` is required.
```

Data fetching with Mongoose

Fetching data from MongoDB collections using Mongoose involves querying the collections using models.

For example, Let's fetch all users from the users' collection
index.js

```
// fetch user data
const fetchUser = async () => {
  try {
    // Fetch all users from the database
    const users = await User.find({});
    console.log('Users:', users);
  } catch (error) {
    console.error('Error fetching users:', error);
  }
};
```

Update Data with Mongoose

Updating data in MongoDB collections using Mongoose involves finding documents and applying update operations.

For example, let's update the user's email address -
index.js

```
// update user
const updateUser = async () => {
  try {
    const result = await User.updateOne(
      { username: 'john_doe' },
      { email: 'john@example.org' }
    );
    console.log('Update result:', result);
  } catch (error) {
    console.error('Error updating user:', error);
  }};
```

```
[nodemon] starting `node index.js`
Connected to MongoDB
Update result: {
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 1
}
```

The above image output shows that it has been updated successfully.

```
[nodemon] restarting due to changes...
Connected to MongoDB
User saved successfully: {
  username: 'john_doe',
  email: 'john@example.com',
  password: 'password123',
  _id: new ObjectId('65eb3252965c04ec948fdf8c'),
  __v: 0
}
```

→

```
Connected to MongoDB
Users: [
  {
    _id: new ObjectId('65eb3252965c04ec948fdf8c'),
    username: 'john_doe',
    email: 'john@example.org',
    password: 'password123',
    __v: 0
  }
]
[nodemon] restarting due to changes...
```

Delete Data with Mongoose

Deleting data from MongoDB collections using Mongoose involves finding documents and removing them from the collection.

For example, Let's delete a user from the 'users' collection:
index.js

```
// delete user
const deleteUser = async () => {
  try {
    const result = await User.deleteOne({ username: 'john_doe' });
    console.log('User deleted successfully');
  } catch (error) {
    console.error('Error deleting user:', error);
  }
};
```

Output -

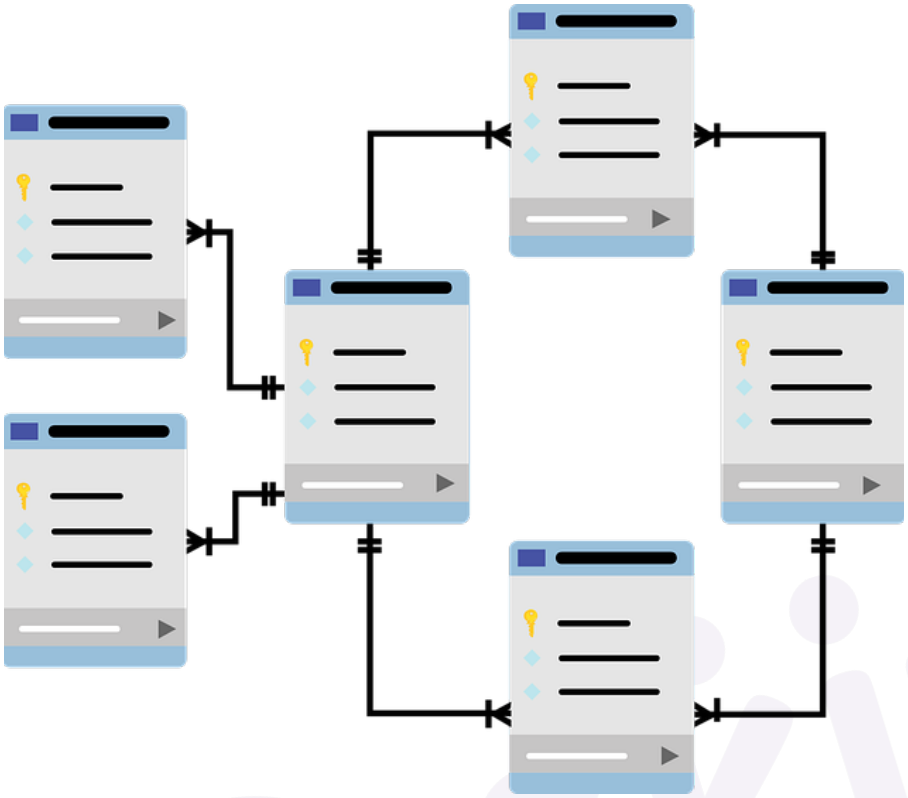
```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Connected to MongoDB
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Connected to MongoDB
User deleted successfully
```

Here is the full code of index.js

```
const mongoose = require('mongoose');
const User = require('./model/users.model');

// Connection URL
const MONGODB_URI = 'mongodb://127.0.0.1:27017/db';

// Connect to MongoDB
mongoose.connect(MONGODB_URI, {});

// Handle connection events
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'Connection error:'));
db.once('open', () => {
  console.log('Connected to MongoDB');
  // createUser(); //un-comment to add user
  // fetchUser(); // un-comment to fetch user data
  // updateUser(); // un-comment to update user data
  // deleteUser(); // un-comment to delete user data
});

// createUser
```

```
const createUser = async () => {
  try {
    // Create a new user instance
    const newUser = new User({
      email: 'john@example.com',
      password: 'password123',
    });

    // Save the user to the database
    const user = await newUser.save();
    console.log('User saved successfully:', user);
  } catch (error) {
    console.error(`User failed to save: ${error}`);
  }
};

// fetch user data
const fetchUser = async () => {
  try {
    // Fetch all users from the database
    const users = await User.find({});
    console.log('Users:', users);
  } catch (error) {
    console.error('Error fetching users:', error);
  }
};

// update user
const updateUser = async () => {
  try {
    const result = await User.updateOne(
      { username: 'john_doe' },
      { email: 'john@example.org' }
    );
    console.log('Update result:', result);
  } catch (error) {
    console.error('Error updating user:', error);
  }
};

// delete user
const deleteUser = async () => {
  try {
    const result = await User.deleteOne({ username: 'john_doe' });
    console.log('User deleted successfully');
  } catch (error) {
    console.error('Error deleting user:', error);
  }
};
```

Relationship



Establishing relationships between MongoDB collections using Mongoose allows you to model complex data structures and build relational databases.

In Mongoose, relationships between two entities or schemas can be established using the following methods:

- Embedding or nesting collections
- Referencing

Embedding or nesting collections in Mongoose involves storing one schema within another schema. This approach is suitable when the nested data is a part of the parent document and doesn't exist independently. In contrast, referencing involves storing a reference to a document from another collection. This method is suitable when the referenced data can exist independently and is shared among multiple documents.

Embedding or nesting collections

In embedding or nesting collections, one schema is embedded within another schema.

For example, in a user and address scenario, the address schema can be embedded within the user schema if each user has a single address associated with them.

i.e

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

const addressSchema = new mongoose.Schema({
  street: String,
  city: String,
  state: String,
  country: String
});

const userSchema = new mongoose.Schema({
  username: String,
  email: String,
  password: String,
  address: addressSchema /*Embedding address schema within user
schema*/
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```

From the above example, the addressSchema is embedded within the userSchema. Each user document will contain the address details within itself.

Referencing

In referencing, documents from one collection reference documents from another collection using their ObjectId. In the user and address scenario, if each user can have multiple addresses and addresses can exist independently, it's more appropriate to reference the address documents from the user documents.

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

const addressSchema = new mongoose.Schema({
  street: String,
  city: String,
  state: String,
  country: String
});

const userSchema = new mongoose.Schema({
  username: String,
  email: String,
  password: String,
  addresses: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Address' }] /*
Referencing address schema */
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```

From the above example, the addresses field in the userSchema is an array of ObjectId references to documents in the Address collection. Each user document stores references to the addresses associated with that user.

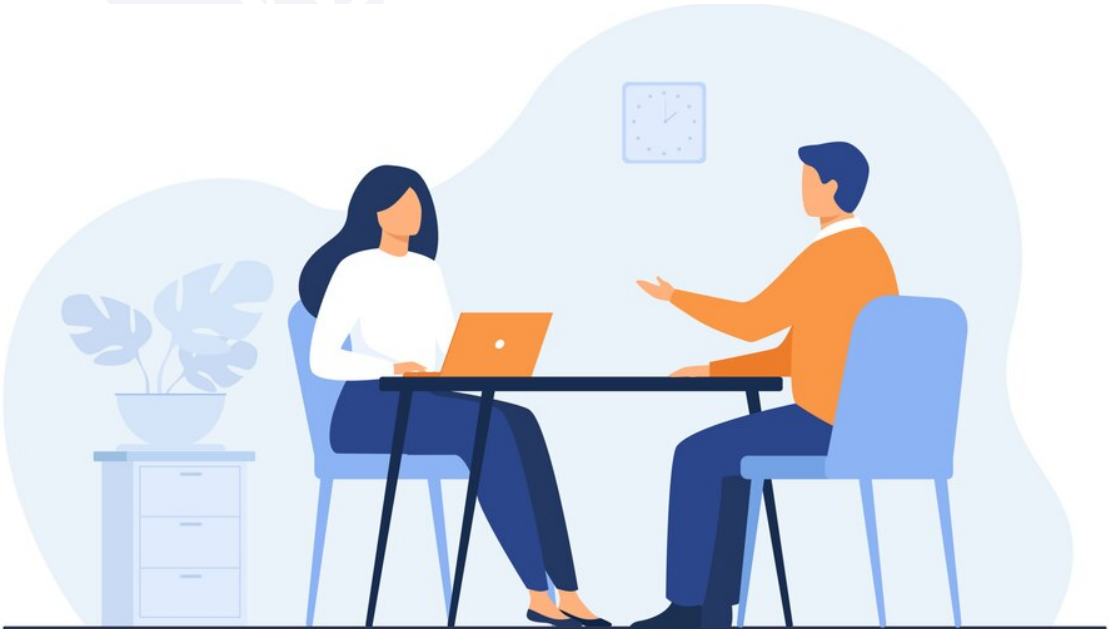
Illustrative example, Suppose we have a user with embedded address data

```
{
  username: 'john_doe',
  email: 'john@example.com',
  password: 'password123',
  address: {
    street: '123 Main St',
    city: 'New York',
    state: 'NY',
    country: 'USA'
  }
}
```

And another user with referenced addresses:

```
{
  username: 'jane_smith',
  email: 'jane@example.com',
  password: 'password456',
  addresses: [ ObjectId("addressId1"), ObjectId("addressId2") ]
}
```

Interview Points



Question 1 - How do you establish a connection to a MongoDB database using Mongoose in a Node.js application, and why is it important to handle connection events and also provide the code example?

Solution -
To establish a connection to a MongoDB database, we use the **mongoose.connect()** method, passing the connection URL and options (optional) as parameters.
It's important to handle connection events such as 'error' and 'open' to ensure robustness and reliability in our application. Handling 'error' events allows us to catch any connection errors and handle them gracefully while handling the 'open' event notifies us when the connection to the database is successfully established.

Here is the code example -

```
// Connection URL
const MONGODB_URI = 'mongodb://127.0.0.1:27017/db';

// Connect to MongoDB
mongoose.connect(MONGODB_URI, {});

// Handle connection events
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'Connection error:'));
db.once('open', () => {
  console.log('Connected to MongoDB');
});
```

Question 2 - Explain the differences between embedding/nesting collections and referencing in Mongoose, and discuss the scenarios where each approach is most suitable.

Solution -
Embedding/nesting collections involves storing one schema within another schema while referencing involves storing references to documents from another collection using their ObjectId.
Embedding is suitable when the nested data is a part of the parent document and doesn't exist independently, resulting in better performance for read operations.
Referencing is suitable when the referenced data can exist independently and is shared among multiple documents, allowing for better data consistency and scalability. It's important to consider factors such as data size, data access patterns, and data consistency requirements when choosing between embedding and referencing in Mongoose.

PW SKILLS

THANK YOU !