BACKEND: Routes, Schema and Models, Controllers

martedì 28 novembre 2023 18:01

LIST OF DEPENDENCIES:
Express: This is a library that simplifies the creation of APIs and server in Node.js
Mongoose: This is a tool that facilitates interaction with MongoDB databases, allowing us to create models, save data, retrieve data, and more.
Cors: This stands for Cross-Origin Resource Sharing, and it aids in communication with the frontend.
Dotenv: This assists with handling .env files, enabling us to create environment variables and securely store confidential data and information.
Nodemon: A utility that auto-restarts the server upon detecting changes.
Bcrypt: it is a cryptographic library that's used for secure password hashing. It helps to safely store password in your application's backend.
Jsonwebtoken: JSON Web Token (JWT) is a secure method for transmitting information between two parties in an encoded JSON format, often used for authentication and authorization.
Validator: it is a library full of strings validators and sanitizers. It's used to validate and sanitize strings input, ensuring data integrity and security.

MongoDB and Mongoose

Q. What is a cluster?
A. A cluster is a group of databases working together.

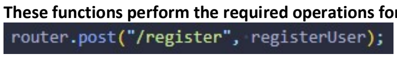
After you create a cluster on MongoDB, copy and then paste the link in .env file:
ATLAS_URI=mongodb+srv://admin:admin@friendzoneapp.1tv0qhe.mongodb.net**/**?retryWrites=true&w=majority

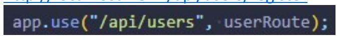
How does this work?
Schema: a schema in MongoDB defines the structure of documents within a collection, specifying fields, their types and validation rules.
Model: a model in MongoDB is a class with which interact with the database (read, insert, update, etc.), based on the defined schema.

Node.js and Express.js

1. HOW TO DEFINE "ROUTES"

After setting up our Models, let's now take care of Routes.

1. **Define Routes:** create a file within the **"Routes"** folder, such as **"userRoute.js"**. This file will contain various HTTP request methods such as "POST" and "GET" if you need them.
   For each method, **define a path** (e.g., **"/register"**) and **associate it with a function** from the "userController" file that you're going to import in your userRoute.js file.
   These functions perform the required operations for each path. In this case it will perform a registration operation.
   
   ```
   router.post("/register", registerUser);
   ```

2. **Use Routes:** In the **"index.js"** file (our server), use the defined routes with the help of Express's **app.use()** method.
   For example, **app.use("/api/users", userRoute);** will use the routes defined in the **"userRoute.js"** file whenever a request is made to **"/api/users".** So the final path to navigate to "/register" will be:
   http://localhost:PORT/api/users/register
   
   ```
   app.use("/api/users", userRoute);
   ```

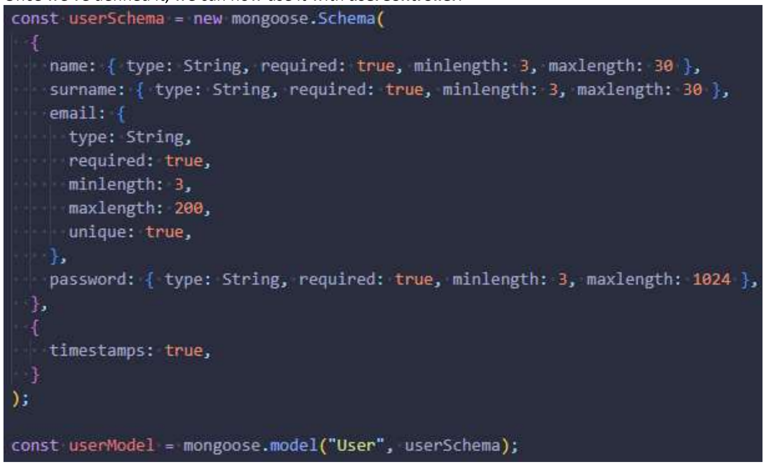
3. **Route Execution:** When a request is made to a specific path, the corresponding function in the **"userController"** file is invoked. For instance, if a POST request is made to **"/api/users/registration"**, then the **registerUser** function is called from the userController file (you need to import the functions first) that will contain all the functions we created.

To summarize, in the **index.js** file (our server), we will find the line **"app.use("/api/users", userRoute);".**
This line will call various paths we defined in our **userRoute.js** file (for example: **"router.post("/register", registerUser);".**
In turn, this will call the **registerUser function** we defined in the **userController.js** file.

2. HOW TO DEFINE CONTROLLERS AND MODELS

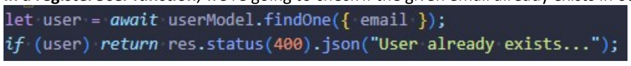
Okay, but what's is missing? We've talked about "userController.js" file, but what is it?
We're going to make use of **"userController.js" file** to define all the functions (e.g., the logic of a registration path) that then we will pass as a second parameter in userRoute.js file when we define a path.
Example: **router.post("/desiredPath", userControllerLogicFunction);**

When setting up controllers, we also make use of the models.
This is our User model and its Schema. We can tell that the name, surname, email and password are required and have a min. length.
Once we've defined it, we can now use it with **userController**.



```
const userSchema = new mongoose.Schema(
  {
    name: { type: String, required: true, minlength: 3, maxlength: 30 },
    surname: { type: String, required: true, minlength: 3, maxlength: 30 },
    email: {
      type: String,
      required: true,
      minlength: 3,
      maxlength: 200,
      unique: true,
    },
    password: { type: String, required: true, minlength: 3, maxlength: 1024 },
  },
  {
    timestamps: true,
  }
);

const userModel = mongoose.model("User", userSchema);
```

Let's say we're still working on **userController,** then we need to import **userModel.** But WHY? Let's see it together.
In a registerUser function, we're going to check if the given email already exists in our database, that's how we can do it with **userModel.findOne({email});**



```
let user = await userModel.findOne({ email });
if (user) return res.status(400).json("User already exists...");
```

Then, we're going to make use of **Validator** to validate our input form data.

```
if (!validator.isEmail(email))
  return res.status(400).json("Email must be a valid email...");

if (!validator.isStrongPassword(password))
  return res.status(400).json("Password must be a strong password..");
```

If every check went fine, now it's time to save our new registered User. This is how we can do it by using our **userModel:**

```
user = new userModel({ name, surname, email, password });
```

But this is not all, we first need to hash the user password by using **Bcrypt**:

```
const salt = await bcrypt.genSalt(10);
user.password = await bcrypt.hash(user.password, salt);
```

Bcrypt.genSalt will create a random hash string of 10 characters

Now it's finally time to save the user in the database with the hashed password:

```
await user.save();
```

Be aware that this will also create an **"_id". Why?** MongoDB add an id when a new document is saved to the database. This happens after you execute this line. The _id will then be accessible via **"user._id"** after the document has been saved.

We also need a JWT token to assign to the user.
Let's go to the **.env** file and add this string: **"JWT_SECRET_KEY = addHereRandomString"**
Now let's go back to our **"userController.js"** file and outside the functions, we write this a function to generate a JWT token:

```
const createToken = (_id) => {
  const jwtSecretKey = process.env.JWT_SECRET_KEY;

  return jwt.sign({ _id }, jwtSecretKey, { expiresIn: "3d" });
};
```

This function takes an id as a parameter (the id of the user), then grabs the jwtSecretKey we defined earlier in our .env file and use it to create a jwt token that will expire in 3 days.
PS: jwt.sig takes up to 3 parameters: the id of the user, jwt secret key (in .env file), and after how many days you want it to be expired.

Once we save the user in the database, we can create the token calling the function "createToken" we just created in the "registerUser" function:

```
const token = createToken(user._id);
```

Let's now send the data to the client. We're going to send the user id, name, surname, email and token. (the pw is private)

```
res.status(200).json({ _id: user._id, name, surname, email, token });
```

You can test this out by sending a post request to the specific path created which will be : "/api/users/register".
Select Body, raw and JSON. Then compile the JSON with the data of the user.
You'll receive as a response the id, name, surname and token.
Once you open MongoDB, in collection/users you'll see a new user.

3. CREATE API FOR CHATS (MODELS, SCHEMA AND ROUTES)

MODELS AND SCHEMA

First of all, just like how we did with users, we need to create a chat model.
So we go in the Models folder and create a file called **"chatModel.js"** and define the Schema for the private chat.

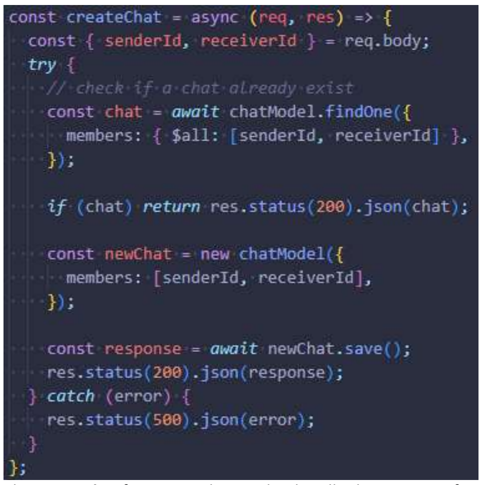
```
const mongoose = require("mongoose");

const chatSchema = new mongoose.Schema(
  {
    members: Array,
  },
  {
    timestamps: true,
  }
);

const chatModel = mongoose.model("Chat", chatSchema);

module.exports = chatModel;
```

So now we know that the chat will be an array of members (the sender and the receiver), and we'll also have a timestamp of when the chat has been created.
We'll use this chatSchema to create our chatModel.

Once we're done with this, we need to create the chatController.
We'll have:

1. **createChat**
2. **userChats**
3. **findChat**

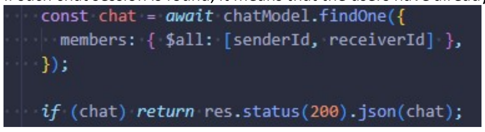
CREATE CHAT



```
const createChat = async (req, res) => {
  const { senderId, receiverId } = req.body;
  try {
    // check if a chat already exist
    const chat = await chatModel.findOne({
      members: { $all: [senderId, receiverId] },
    });

    if (chat) return res.status(200).json(chat);

    const newChat = new chatModel({
      members: [senderId, receiverId],
    });

    const response = await newChat.save();
    res.status(200).json(response);
  } catch (error) {
    res.status(500).json(error);
  }
};
```

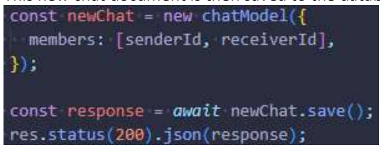
The **createChat** function is designed to handle the creating of a new chat session between two users.
When creating a chat, we'll be having two IDs in the request body: the **senderId** and **receiverId,** so we're going to extract them from **req.body** (it will contain the IDs of the two user who are having a conversation).

Next, the function performs a check to see if a chat session already exists between these two users.
This is done using the **chatModel.findOne** method, which searches for a chat document where the **members** array contains both the **senderId** and **receiverId.**
The **$all** operator is used here to ensure that the document returned must include all specified **IDs** in the array, regardless of their order or presence of additional users.
If such chat session is found, it means that the users have already established a chat, and the existing chat document is returned with a **200 status code.**



```
    const chat = await chatModel.findOne({
      members: { $all: [senderId, receiverId] },
    });

    if (chat) return res.status(200).json(chat);
```

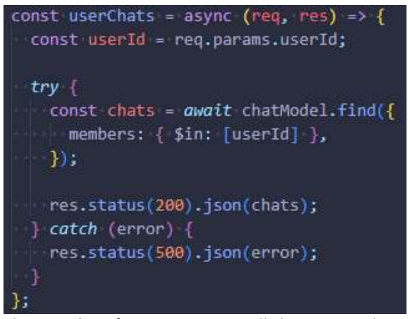
However, if no existing chat is found, a new chat document is created with the **chatModel** constructor, where the **members** array is populated with the **senderId** and **receiverId**.
This new chat document is then saved to the database, and the newly created chat session is returned to the user (frontend) with a **200** status code.



```
const newChat = new chatModel({
  members: [senderId, receiverId],
});

const response = await newChat.save();
res.status(200).json(response);
```

In case of any errors during this process, such as issues with the database operation, the function will catch the error and responds with a **500 status code**, indicating an internal server error.

USER CHATS



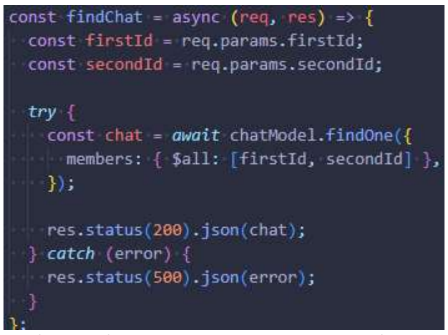
```
const userChats = async (req, res) => {
  const userId = req.params.userId;

  try {
    const chats = await chatModel.find({
      members: { $in: [userId] },
    });

    res.status(200).json(chats);
  } catch (error) {
    res.status(500).json(error);
  }
};
```

The **userChats** function retrieves all chat session that a specific user is part of.
It works by taking the current logged in user's ID from the URL parameters (**req.params.userId)**, and then uses the **chatModel.find** method to search the database for any chat documents where the user's ID appears in the **members** array.

The **$in** MongoDB operator is used here to find documents where the specified value (the user's ID) is in the given array (the **members** array of the chat document).
This means that any chat that includes the user, either as a sender or receiver, will be returned.

If the search is succesful, the function sends back a list of all the chats involving the user with a **200 status code**, allowing the user to see all their ongoing conversations.

FIND CHAT



```
const findChat = async (req, res) => {
  const firstId = req.params.firstId;
  const secondId = req.params.secondId;

  try {
    const chat = await chatModel.findOne({
      members: { $all: [firstId, secondId] },
    });

    res.status(200).json(chat);
  } catch (error) {
    res.status(500).json(error);
  }
};
```

The **findChat** function is designed to locate a specific **chat session** between **two users.**
This function is crucial for enabling the users to retrieve their existing conversation with another user without creating a duplicate chat session.

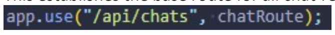
It starts by retrieving the **IDs** of these users from the URL parameters **(firstId** and **secondId).**
With these **IDs,** it uses the **chatModel.findOne** method to search the database for a chat document that includes **both user IDs** in its **members** array.

The **$all** operator ensures that the returned document must contain **all** the specified **IDs**, meaning the function is looking for a chat where both users are participants.

If such a chat is found, the function responds with the chat data and a **200 status code,** indicating a successful operation.

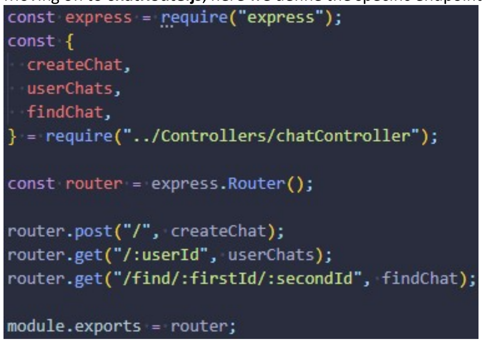
NOW THAT WE HAVE CREATED THE CONTROLLERS FOR THE CHAT, WE ALSO NEED TO CREATE THE ROUTES.

In the **index.js** file, we're setting up the server specifying that any request to **/api/chats** should be handled by **chatRoute.js (which is a middleware).**
This establishes the base route for all chat-related endpoints.



```
app.use("/api/chats", chatRoute);
```

Moving on to **chatRoute.js**, here we define the specific endpoints for chat operations using the Express router.



```
const express = require("express");
const {
  createChat,
  userChats,
  findChat,
} = require("../Controllers/chatController");

const router = express.Router();

router.post("/", createChat);
router.get("/:userId", userChats);
router.get("/find/:firstId/:secondId", findChat);

module.exports = router;
```

Here we map:

1. the **HTTP POST** request to the root path **" / "** and let it be handled by the **createChat** middleware controller function
2. the **HTTP GET** request to the **/:userId** path and let it be handled by the **userChat** middleware controller function
3. the **HTTP GET** request to the **/find/:firstId/:secondId** path and let it be handled by the **findChat** middleware controller function

All these middleware controller functions are imported from **chatController.js** file.

In **chatController.js,** the chat operations are implemented like this:

1. The **createChat** function checks for an existing chat between two users and creates a new one if necessary.
2. The **userChats** function retrieves all chats that include a specific user.
3. The **findChat** function locates a chat between two specific users.

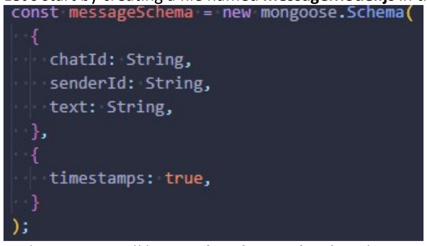
All functions interact with the **chatModel** which represents the chat data structure in the database.

The **index.js** file directs incoming chat requests to the appropriate controller functions through the routes defined in **chatRoute.js**.

4. CREATE API FOR MESSAGES (MODELS, SCHEMA AND ROUTES)

In the last API we created, the **Chat API**, we were able to create a chat, list all the chats and find a specific chat.
Now, we're about to create our **Message API** so that we will be able to manage messages related to chats.

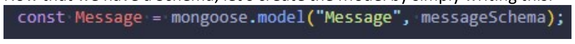
Let's start by creating a file named **messageModel.js** in the Models folder and define a Schema for it.



```
const messageSchema = new mongoose.Schema(
  {
    chatId: String,
    senderId: String,
    text: String,
  },
  {
    timestamps: true,
  }
);
```

Each message, will have a **chatId,** a **senderId**, and a **text**

Now that we have a Schema, let's create the Model by simply writing this:

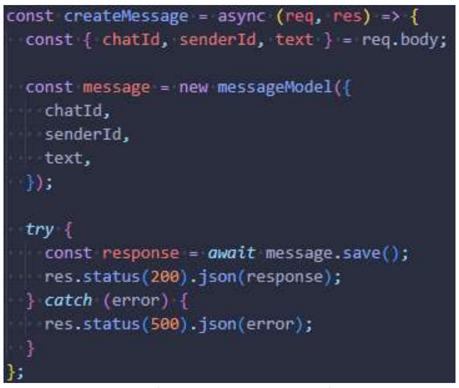


```
const Message = mongoose.model("Message", messageSchema);
```

Once we're done with this, we need to create the messageController.
We'll have:

1. **createMessage**
2. **getMessages**

CREATE MESSAGE



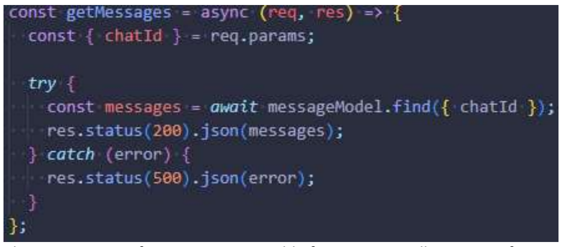
```
const createMessage = async (req, res) => {
  const { chatId, senderId, text } = req.body;

  const message = new messageModel({
    chatId,
    senderId,
    text,
  });

  try {
    const response = await message.save();
    res.status(200).json(response);
  } catch (error) {
    res.status(500).json(error);
  }
};
```

The **createMessage** function is responsible for creating a new message in a chat.
This function is essential for adding news messages to a chat, allowing users to communicate with each other within the application. It ensures that each message is properly stored and associated with the correct chat and sender.

It starts by requiring the **messageModel**.**js**, which is the schema for message data in the database.
The function then extracts the **chatId, senderId** and **text** from the request body, which are: the identifiers for the chat, the sender of the message and the message content, respectively.

A new message document is create using the **messageModel** with the provided details. The function then attempts to **save** this new message to the database. If successful, it sends back the saved message data with a **200 status code**.

GET MESSAGES



```
const getMessages = async (req, res) => {
  const { chatId } = req.params;

  try {
    const messages = await messageModel.find({ chatId });
    res.status(200).json(messages);
  } catch (error) {
    res.status(500).json(error);
  }
};
```

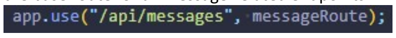
The **getMessages** function is responsible for retrieving all messages from a specific chat in the application.
This function is essential for displaying the conversation history in a chat, allowing users to view previous messages within the application.
It ensures that the messages are properly fetched and presented in the context of their respective chats.

It starts by requiring the **messageModel**, which is the schema for the message data in the database. The function then extracts the **chatId** from the request parameters, which is the identifier for the chat whose messages are to be fetched.

Using **messageModel**, the function performs a database query with the **find** method, passing **{ chatId }** as the query condition.
This retrieves all message documents associated with the given **chatId.**
If the query is succesful, the function sends back the array of retrieved messages with a **200 status code**, indicating that the operation was successful.
In case of any errors during the query operation, such as a validation error or database connection issue, the function will respond with a **500 status code**, which signifies an internal server error.

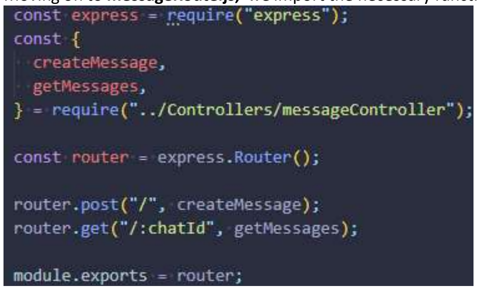
NOW THAT WE HAVE CREATED THE CONTROLLERS FOR THE MESSAGES, WE ALSO NEED TO CREATE THE ROUTES.

In the **index.js file,** we configure the server to handle requests related to messages. Any request that starts with **/api/messages/** is directed to **messageRoute.js**, which is a middleware that defines the base route for all message-related endpoints.



```
app.use("/api/messages", messageRoute);
```

Moving on to **messageRoute.js,** we import the necessary functions from **messageController.js** and use the Express router to define specific endpoints for message operations:



```
const express = require("express");
const {
  createMessage,
  getMessages,
} = require("../Controllers/messageController");

const router = express.Router();

router.post("/", createMessage);
router.get("/:chatId", getMessages);

module.exports = router;
```

Here we map:

1. the **HTTP POST** request to the root path **" / "** to the **createMessage** middleware function, which handles the creation of new messages.
2. the **HTTP GET** request to the **/:chatId** path to the **getMessages** middleware function, which retrieves all messages for a specific chat.
   All these middleware controller functions are imported from **messageController.js** file.

In **messageController.js,** the message operations are implemented like this:

1. The **createMessage** function takes the **chatId, senderId**, and **text** from the request body and creates a new message using the **messageModel.** It then saves this message to the database and returns the saved message data if succesful, or an error if there is a problem.
2. The **getMessages** function uses the **chatId** from the request parameters to find all messages associated with that chat using **messageModel.** It returns these messages if the query is successful, or an error if there is a problem.

The **index.js** file ensures that incoming message requests are directed to the appropriate controller functions through the routes defined in **messageRoute.js**.
This setup allows for clear separation of concerns, with **index.js** handling the routing, **messagesRoute.js** defining the endpoints, and **messageController.js** implementing the logic for message operations.
The **messageModel** represents the message data structure in the database and is used by the controller functions to interact with the database.