FALL SEMESTER (2024-25)

DATABASE SYSTEMS (THEORY)

COURSE CODE: BCSE303L

SLOT:A2+TA2

Assignment Title -

"Chat Application"

Name's : K. Ganesh Maharaj, M.Sai Abhinav Reddy, K.Abhinav, Ch.Poojitha

Reg. No's : 22BDS0168, 22BCT0134, 22BKT0062, 22BCT0313

Slot : A2+TA2(L9+L10)

Video Link : https://drive.google.com/file/d/1XgOEfukKyzb7gsg-WLfLqBIhBSRWF107/view?usp=sharing

GitHub Repo Link : https://github.com/GANESH-MAHARAJ/Syncronus-chat-app

Submitted to

Dr. RAJESHKANNAN R

School of Computer Science and Engineering

VIT, Vellore

Tamil Nadu – 632 014

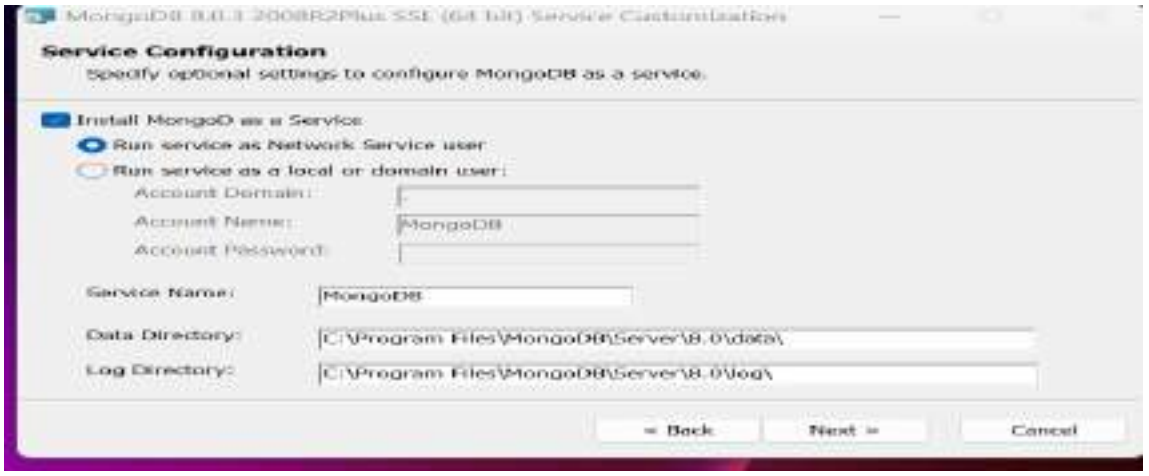
For Windows:

1. **Download MongoDB Installer**
   - Go to MongoDB's download page.
   - Select the **Community Server** version, choose your **Operating System** as Windows, and click **Download**.

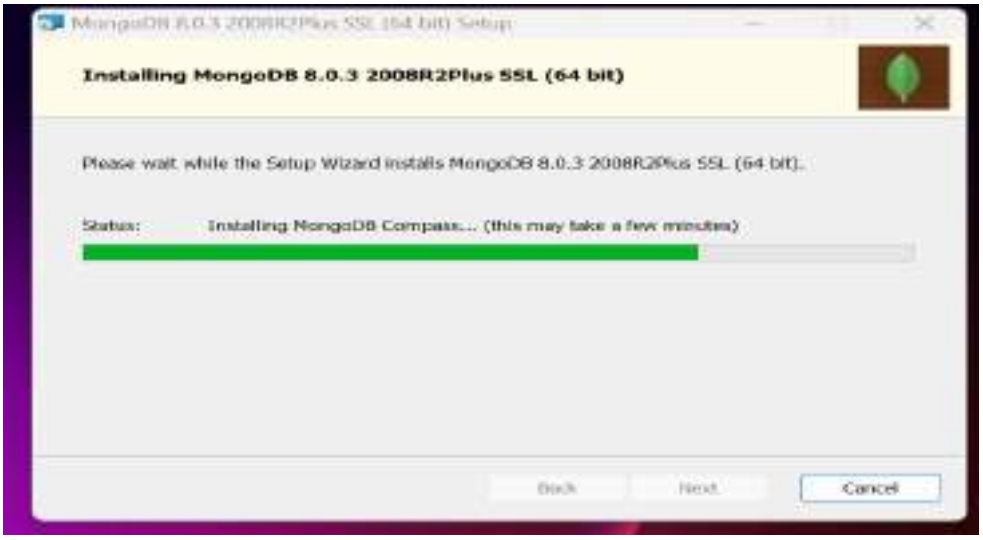
…….

2. **Run the Installer**
   - Open the downloaded .msi file.
   - Follow the setup wizard and select the **Complete** installation option.

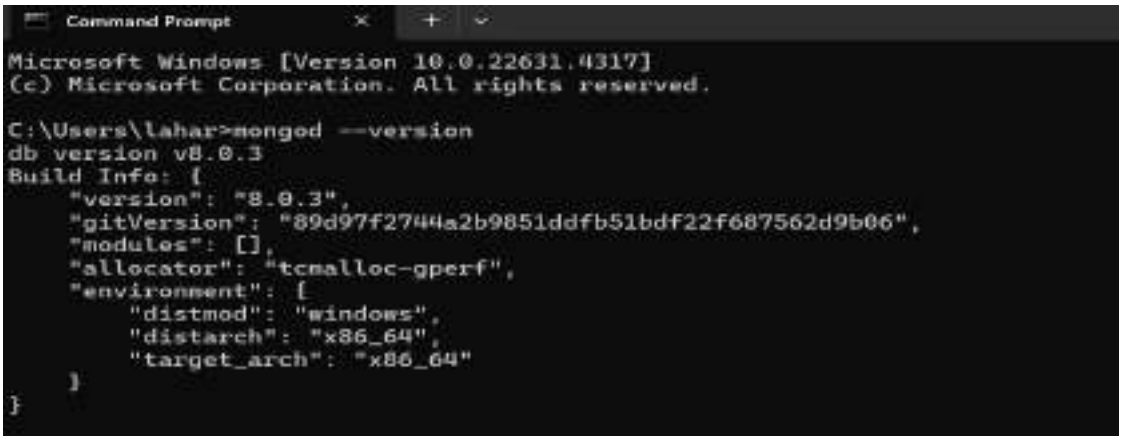
3. **Choose Setup Options**
   - When prompted, select **Install MongoDB as a Service** so MongoDB starts automatically when Windows starts.



   - Choose to install **MongoDB Compass**



4. **Finish Installation**
   - Click **Install** to begin the installation process.
   - After completion, click **Finish**.
5. **Add MongoDB to System Path**
   - To use mongo commands from any directory, add MongoDB's bin folder to your system's PATH.
     - Open the **Environment Variables** (search Edit the system environment variables in Windows).
     - In **System Variables**, find and select Path, then **Edit**.
     - Add the path to your MongoDB bin directory (e.g., C:\Program Files\MongoDB\Server\6.0\bin).
6. **Verify Installation**
   - Open **Command Prompt**.
   - Run mongod --version to confirm MongoDB is installed.

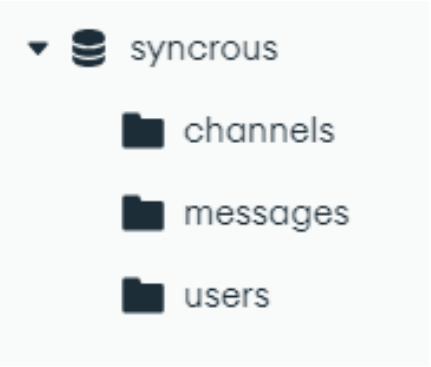


7. **Start MongoDB**
   - In Command Prompt, run mongod to start the MongoDB server.

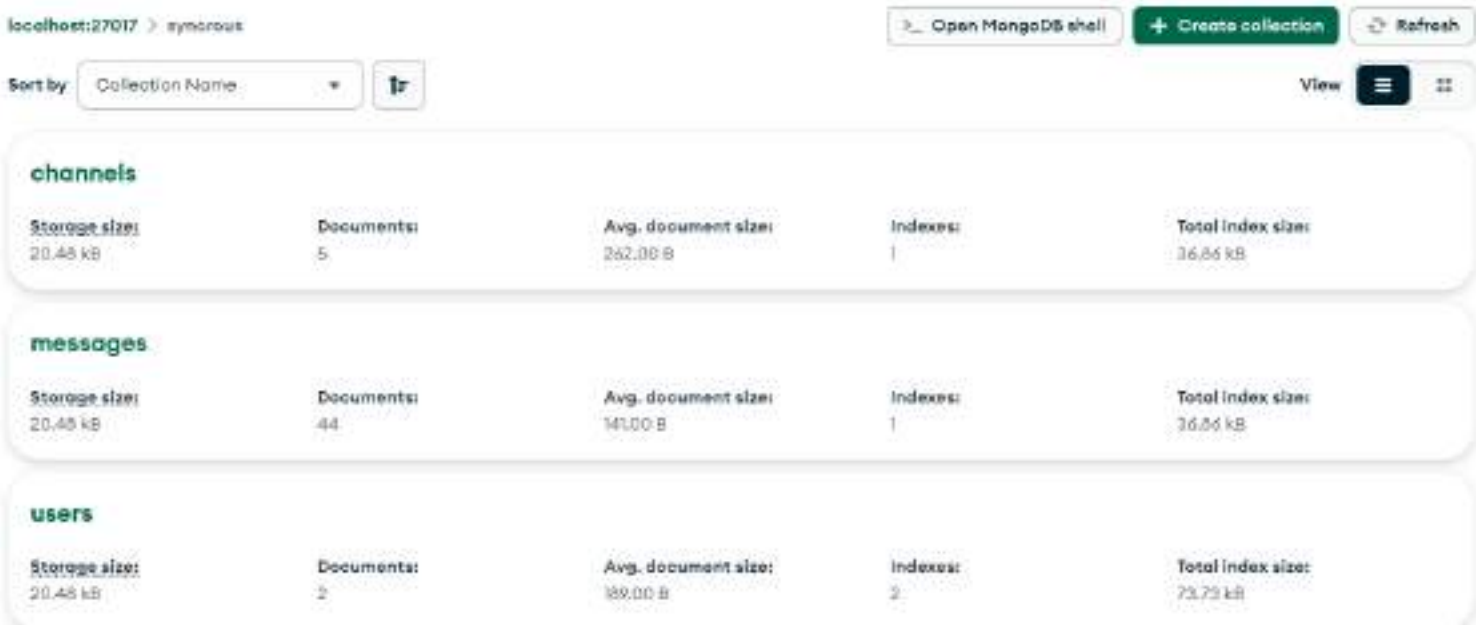
1. Schema Design

Explanation: The schema design involves defining the collections required for the chat application, such as users, messages, and chat rooms.

Collections in Database:



Detailed overview of Collections:



Each collection needs appropriate fields **:**

- **Users** collection: fields like *ID, email, password, profileSetup, color firstName, lastName and image.*

Eg **:**

JS code :

```
JS UserModel.js ×
server > model > JS UserModel.js > (x) userSchema > image
import mongoose from "mongoose";
import bcrypt from "bcrypt";

const userSchema = new mongoose.Schema({
  email: {
    type: String,
    required: [true, "Email is Required"],
    unique: true,
  },
  password: {
    type: String,
    required: [true, "Password is Required"],
  },
  firstName: {
    type: String,
    required: false,
  },
  lastName: {
    type: String,
    required: false,
  },
  image: {
    type: String,
    required: false,
  },
  profileSetup: {
    type: Boolean,
    default: false,
  },
  color: {
    type: Number,
    required: false,
  },
});
```

```
35
36  userSchema.pre("save", async function (next) {
37    const salt = await bcrypt.genSalt();
38    this.password = await bcrypt.hash(this.password, salt);
39    next();
40  });
41
42  userSchema.statics.login = async function (email, password) {
43    const user = await this.findOne({ email });
44    if (user) {
45      const auth = await bcrypt.compare(password, user.password);
46      if (auth) {
47        return user;
48      }
49      throw Error("incorrect password");
50    }
51    throw Error("incorrect email");
52  };
53
54  const User = mongoose.model("Users", userSchema);
55  export default User;
56
```

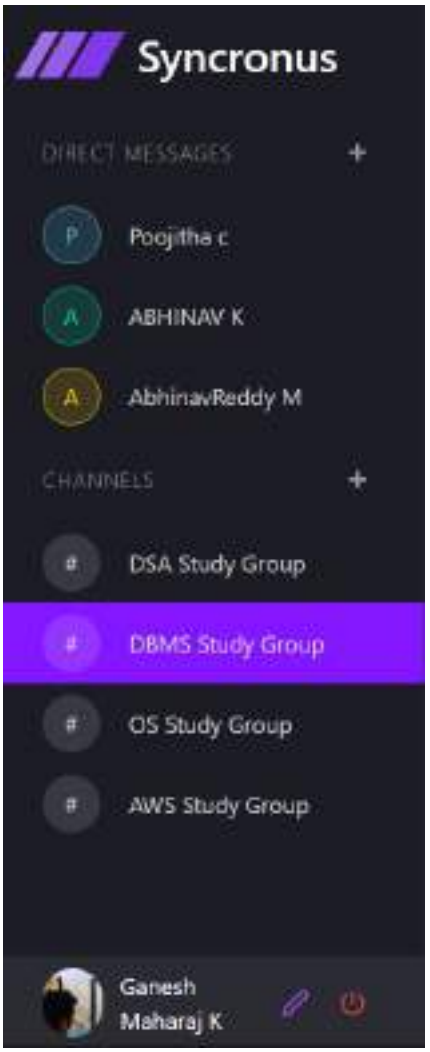
MongoDB User Collection :

```
_id: ObjectId('672ce772a389c7acaf27f470')
email : "ganesh@gmail.com"
password : "$2b$10$UwMddvc/5NdghCynos4JyOOuRxdL3.64T/jOmprJSZms2JBD8t5I6"
profileSetup : true
__v : 0
color : 0
firstName : "Ganesh Maharaj"
lastName : "K"
image : "uploads/profiles/1730996685190profilePic.jpg"


_id: ObjectId('672ce8a8a389c7acaf27f484')
email : "abhinavreddy@gmail.com"
password : "$2b$10$256z2wI2eR3/53O9P7y5bOnkcrUhhpNCWnmixLFW5hZtKJhBu4VX2"
profileSetup : true
__v : 0
color : 1
firstName : "AbhinavReddy"
lastName : "M"
```

```
_id: ObjectId('672ce8f5a389c7acaf27f491')
email : "abhinavkenguva@gmailcom"
password : "$2b$10$hsEdvFFXp485BxvMuvDnFuUC/z.zuge0pNWC9b/zIuv.LsaDZPCJK"
profileSetup : true
__v : 0
color : 2
firstName : "ABHINAV"
lastName : "K"
```

```
_id: ObjectId('672ce96fa389c7acaf27f49a')
email : "poojitha@gmail.com"
password : "$2b$10$HKrl9z5Atl7TZNd7e35OouZpgmYAA/0.7afQoKEoBOXuktXTbFZF."
profileSetup : true
__v : 0
color : 3
firstName : "Poojitha"
lastName : "c"
```


User Collection in Website:



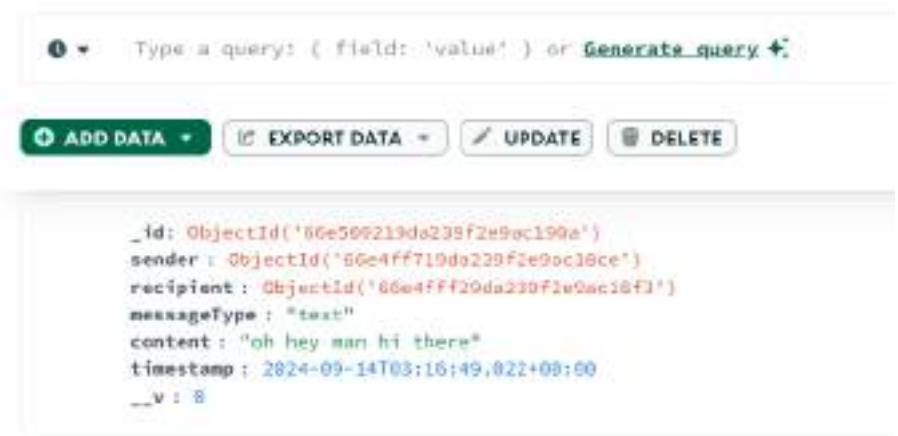
- **Messages** collection **:** fields like messageID, senderID, chatRoomID, content, timestamp.

Eg **:**

JS code :

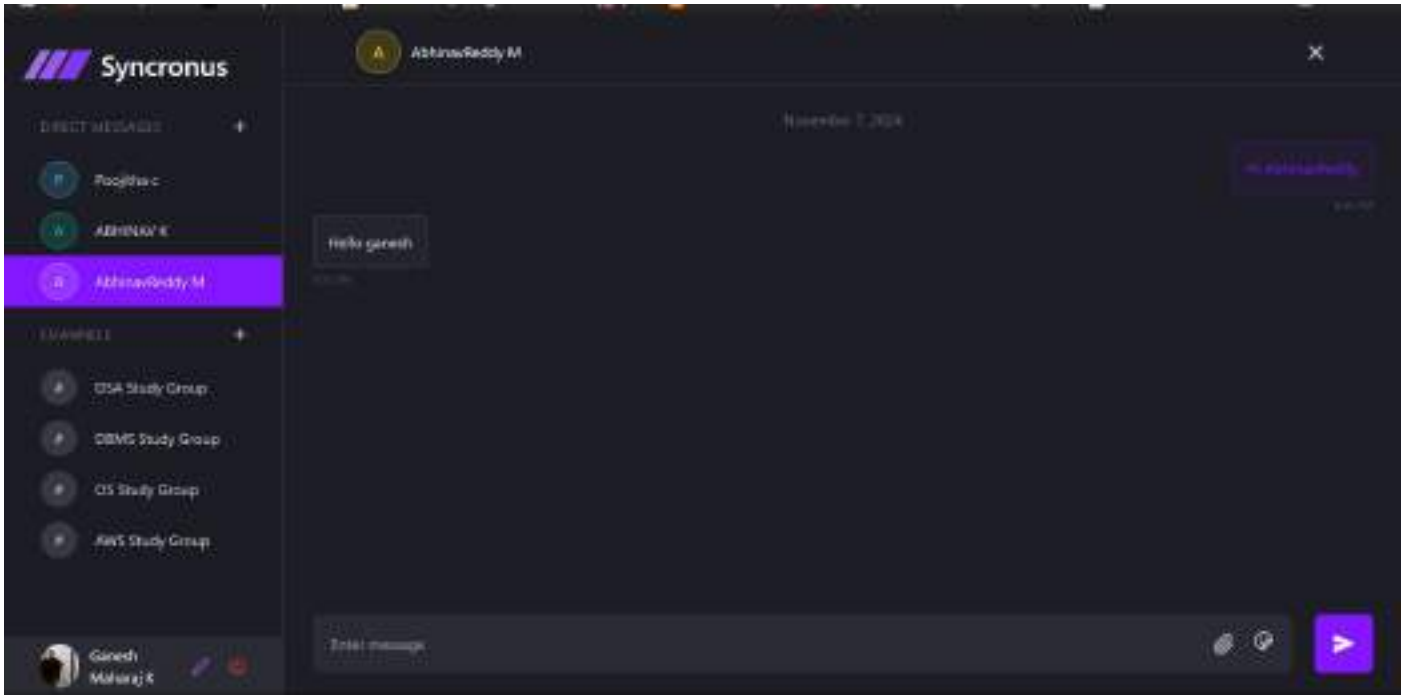
```
JS MessagesModel.js ×
server > model > JS MessagesModel.js > [e] messageSchema > fileUrl
import mongoose from "mongoose";
const messageSchema = new mongoose.Schema({
  sender: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Users",
    required: true,
  },
  recipient: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Users",
    required: false,
  },
  messageType: {
    type: String,
    enum: ["text", "audio", "file"],
    required: true,
  },
  content: {
    type: String,
    required: function () {
      return this.messageType === "text";
    },
  },
  audioUrl: {
    type: String,
    required: function () {
      return this.messageType === "audio";
    },
  },
  fileUrl: {
    type: String,
    required: function () {
      return this.messageType === "file";
    },
  },
  timestamp: {
    type: Date,
    default: Date.now,
  },
});
const Message = mongoose.model("Messages", messageSchema);
export default Message;
```

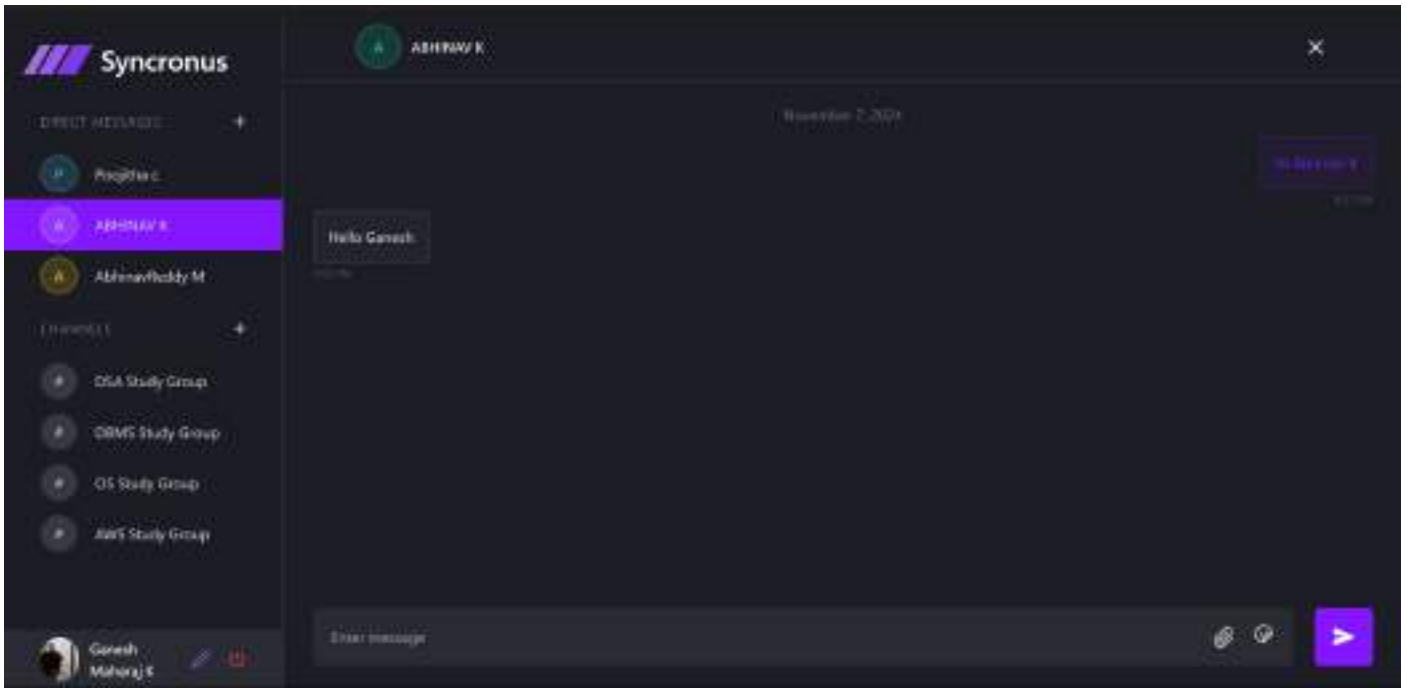
MongoDB Message Collection :



```
_id: ObjectId('672cebe4a389c7acaf27f52d')
sender : ObjectId('672ce772a389c7acaf27f470')
recipient : null
messageType : "text"
content : "Need to Submit ASAP !!! 😊"
timestamp : 2024-11-07T16:33:40.356+00:00
__v : 0
```


Direct messages in Website:





- **Chat Rooms** collection : fields like id, name, members, admin, messages, createdAt and updatedAt

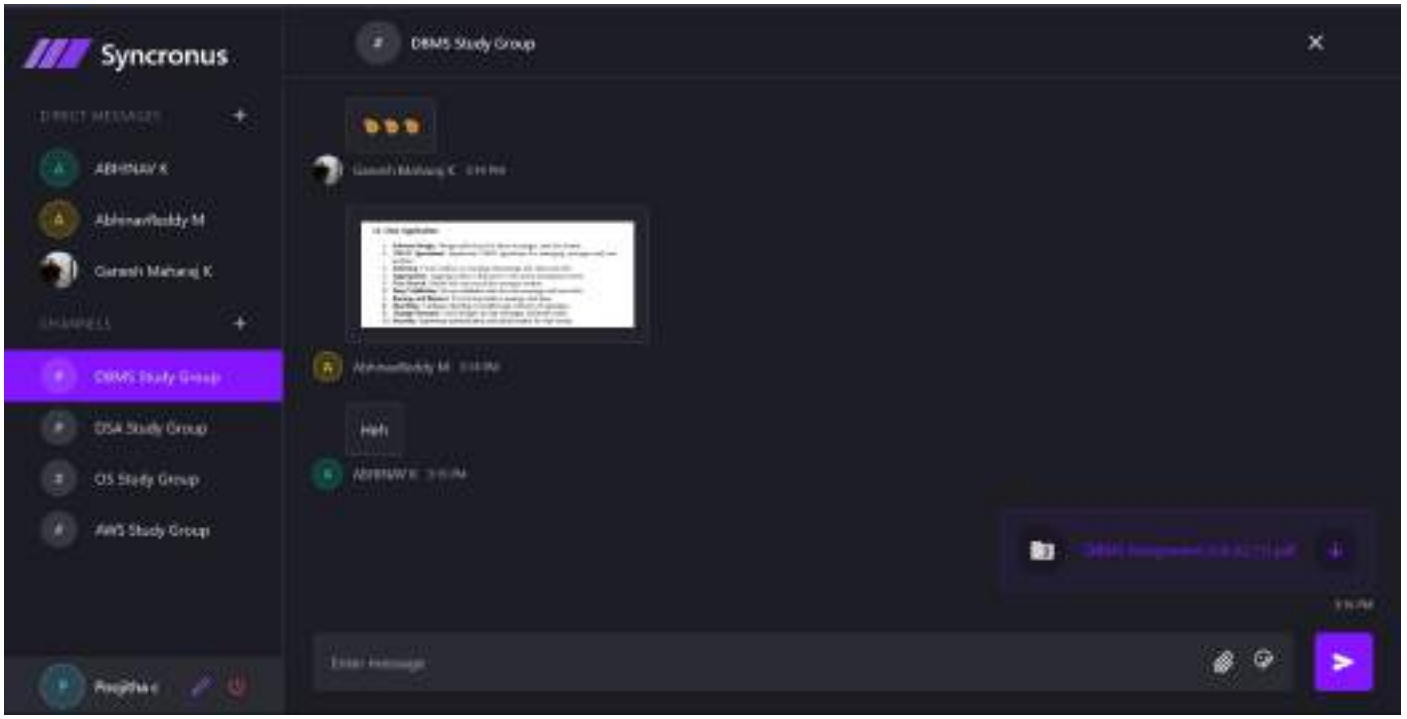
JS code :

JS ChannelModel.js ×

server > model > JS ChannelModel.js > ...

```js
import mongoose from "mongoose";
const channelSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  members: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "Users",
      required: true,
    },
  ],
  admin: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Users",
    required: true,
  },
  messages: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "Messages",
      required: false,
    },
  ],
  createdAt: {
    type: Date,
    default: Date.now,
  },
  updatedAt: {
    type: Date,
    default: Date.now,
  },
});

channelSchema.pre("save", function (next) {
  this.updatedAt = Date.now();
  next();
});

channelSchema.pre("findOneAndUpdate", function (next) {
  this.set({ updatedAt: Date.now() });
  next();
});

const Channel = mongoose.model("Channels", channelSchema);
export default Channel;
```

Chat Room in Website:



MongoDB ChatRooms Collection :

```
_id: ObjectId('672ceb44a389c7acaf27f508')
name : "DBMS Study Group"
▸ members : Array (3)
admin : ObjectId('672ce772a389c7acaf27f470')
▸ messages : Array (19)
createdAt : 2024-11-07T16:31:00.319+00:00
updatedAt : 2024-11-07T16:48:39.431+00:00
__v : 0
```

Questions:

Q1: How did you design the relationships between users, messages, and chat rooms in your schema?

- **Answer**: We designed three main collections: **Users**, **Messages**, and **ChatRooms**. In the Users collection, each user has a unique userID, username, and password hash for authentication. Messages include a senderID (linked to Users) and chatRoomID (linked to ChatRooms). The ChatRooms collection holds chatRoomID, roomName, and an array of memberIDs, creating relationships between users and chat rooms. This structure allows efficient querying for user profiles and messages within specific chat rooms.

Q2: Why did you choose specific fields and data types for each collection?

- **Answer**: We selected fields like userID, messageID, and chatRoomID to uniquely identify each entity, while using String for textual data (like username and content) and Date for timestamp. This setup ensures consistent data storage and efficient retrieval, while the use of Array for members in the ChatRooms collection simplifies participant management.

2. CRUD Operations

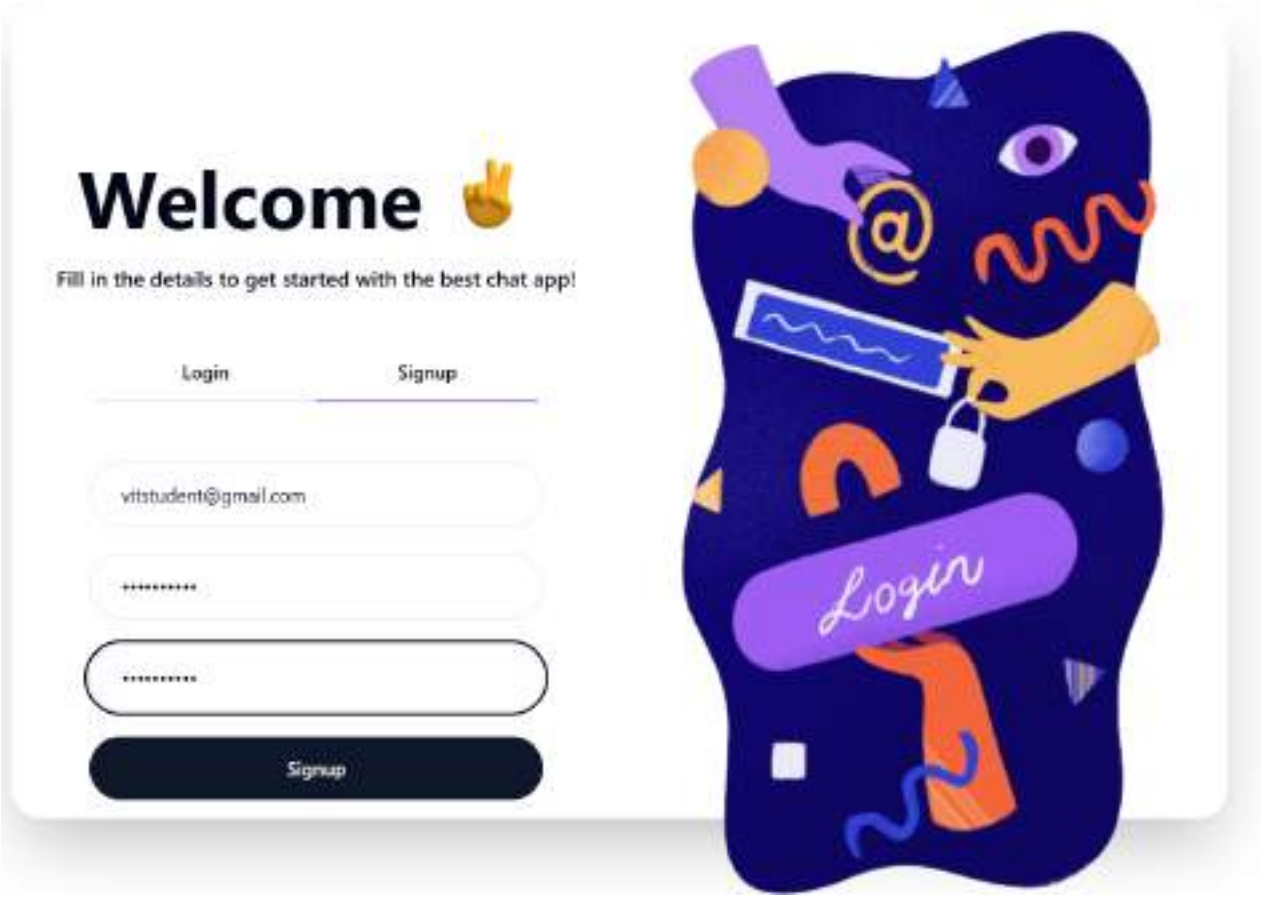
Explanation: CRUD operations (Create, Read, Update, Delete) are essential to manage user profiles, chat rooms, and messages in the application.

- **Create**: Adding registering users, new messages, creating chat rooms.

JS Code for creating new user:

```
export const signup = async (req, res, next) => {
  try {
    const { email, password } = req.body;
    if (email && password) {
      const user = await User.create({ email, password });
      res.cookie("jwt", createToken(email, user.id), {
        maxAge
      });

      return res.status(201).json({
        user: {
          id: user?.id,
          email: user?.email,
          firstName: user.firstName,
          lastName: user.lastName,
          image: user.image,
          profileSetup: user.profileSetup,
        },
      });
    } else {
      return res.status(400).send("Email and Password Required");
    }
  } catch (err) {
    console.log(err);
    return res.status(500).send("Internal Server Error");
  }
};
```

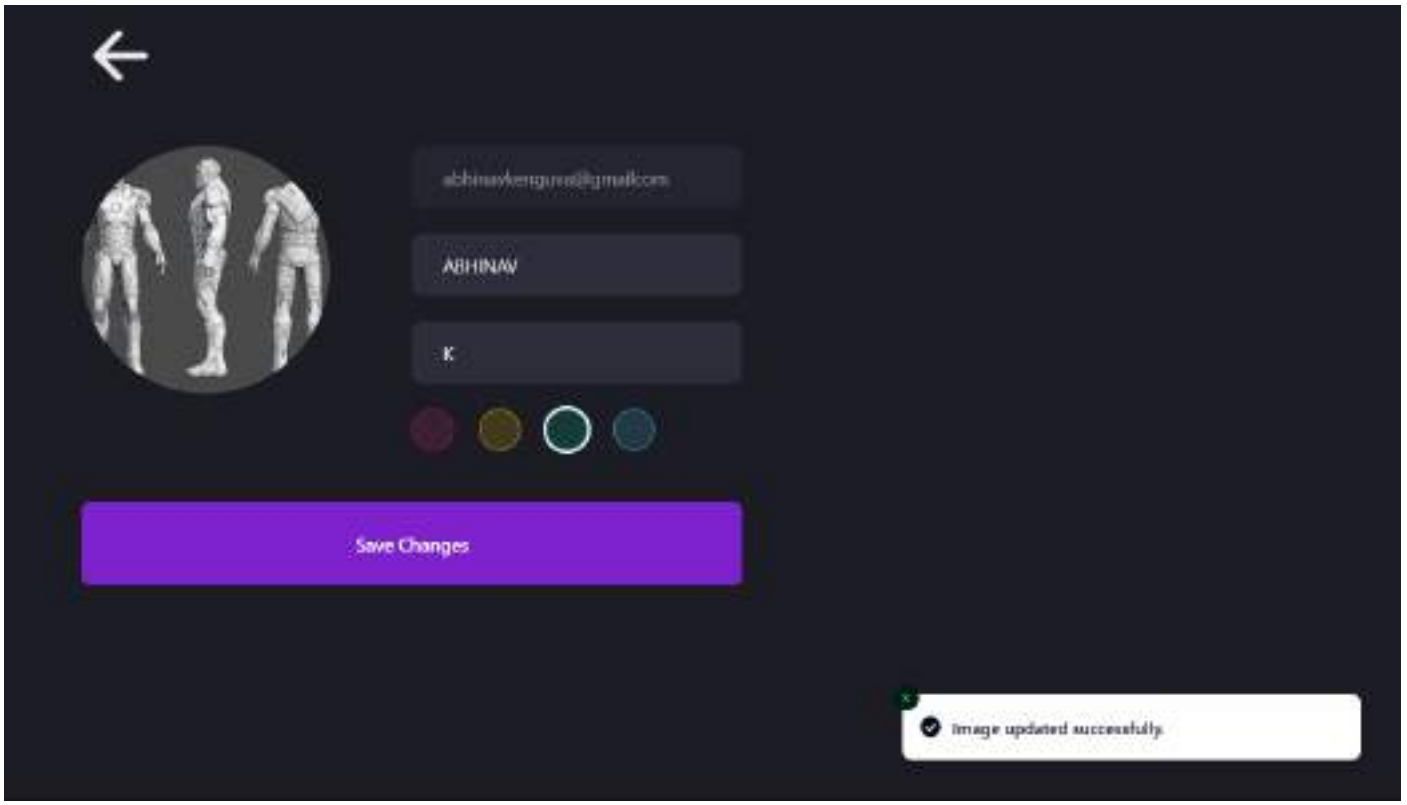
Creating new user(signup page):



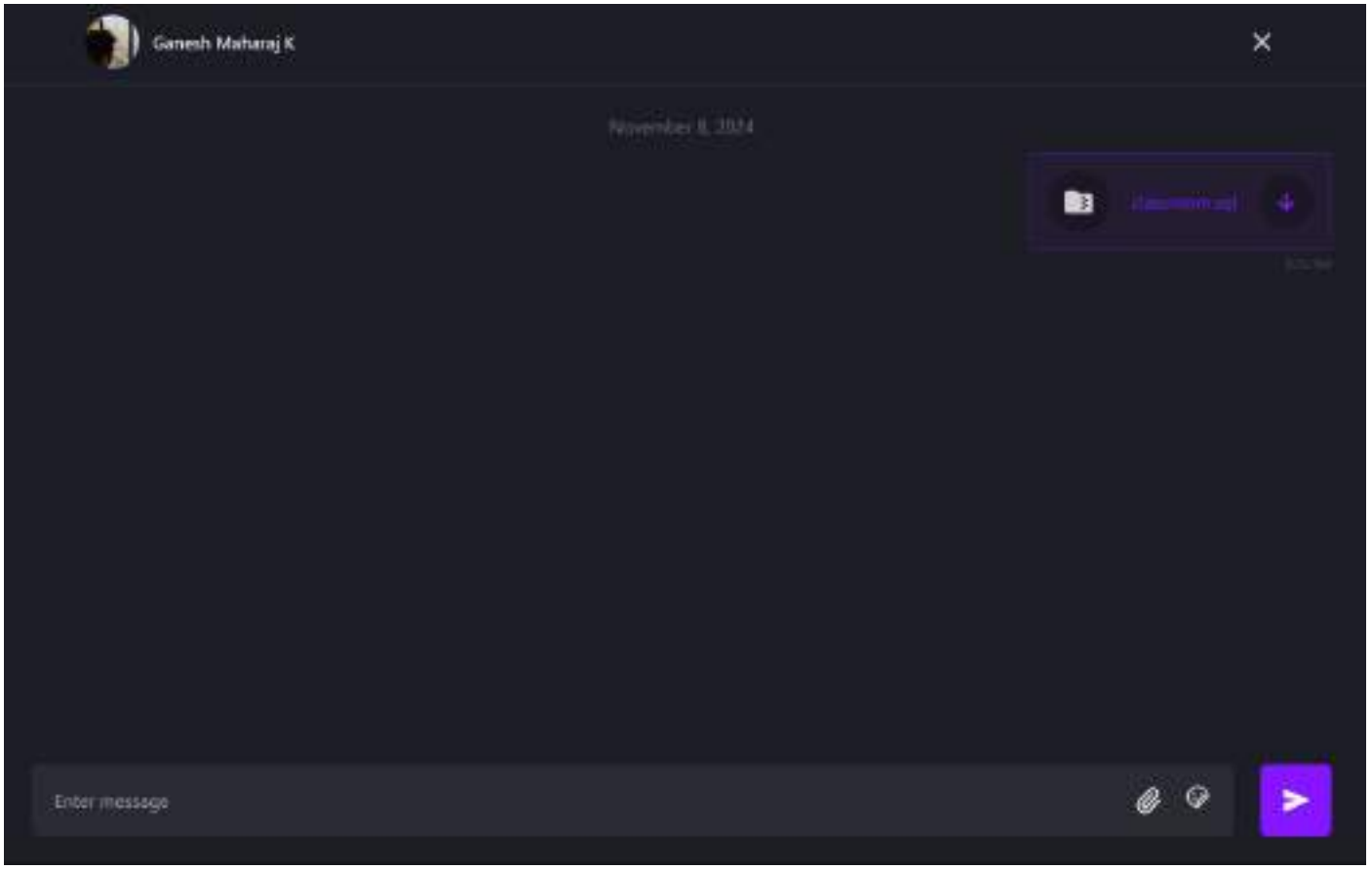
JS Code for uploading User data:

```js
export const uploadFile = async (request, response, next) => {
  try {
    if (request.file) {
      console.log("in try if");
      const date = Date.now();
      let fileDir = `uploads/files/${date}`;
      let fileName = `${fileDir}/${request.file.originalname}`;

      // Create directory if it doesn't exist
      mkdirSync(fileDir, { recursive: true });

      renameSync(request.file.path, fileName);
      return response.status(200).json({ filePath: fileName });
    } else {
      return response.status(404).send("File is required.");
    }
  } catch (error) {
    console.log({ error });
    return response.status(500).send("Internal Server Error.");
  }
};
```

Creating new profile:



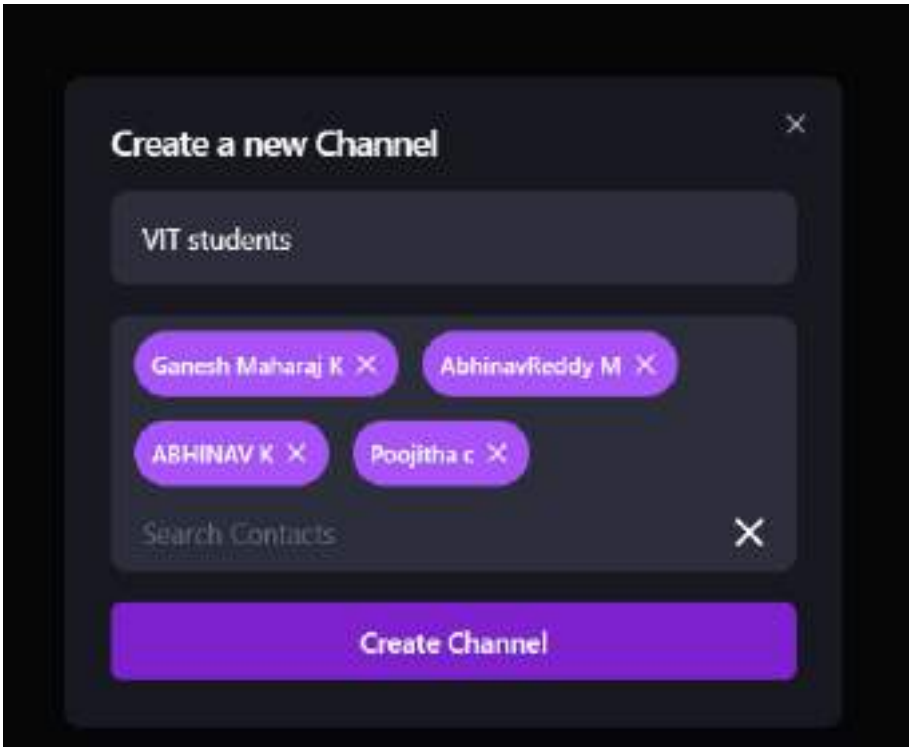
Sending messages/files to another user:



JS Code for creating new channel:

```
export const createChannel = async (request, response, next) => {
  try {
    const { name, members } = request.body;
    const userId = request.userId;
    const admin = await User.findById(userId);
    if (!admin) {
      return response.status(400).json({ message: "Admin user not found." });
    }

    const validMembers = await User.find({ _id: { $in: members } });
    if (validMembers.length !== members.length) {
      return response
        .status(400)
        .json({ message: "Some members are not valid users." });
    }

    const newChannel = new Channel({
      name,
      members,
      admin: userId,
    });

    await newChannel.save();

    return response.status(201).json({ channel: newChannel });
  } catch (error) {
    console.error("Error creating channel:", error);
    return response.status(500).json({ message: "Internal Server Error" });
  }
};
```

Creating a channel and selecting users to be included in the new channel:

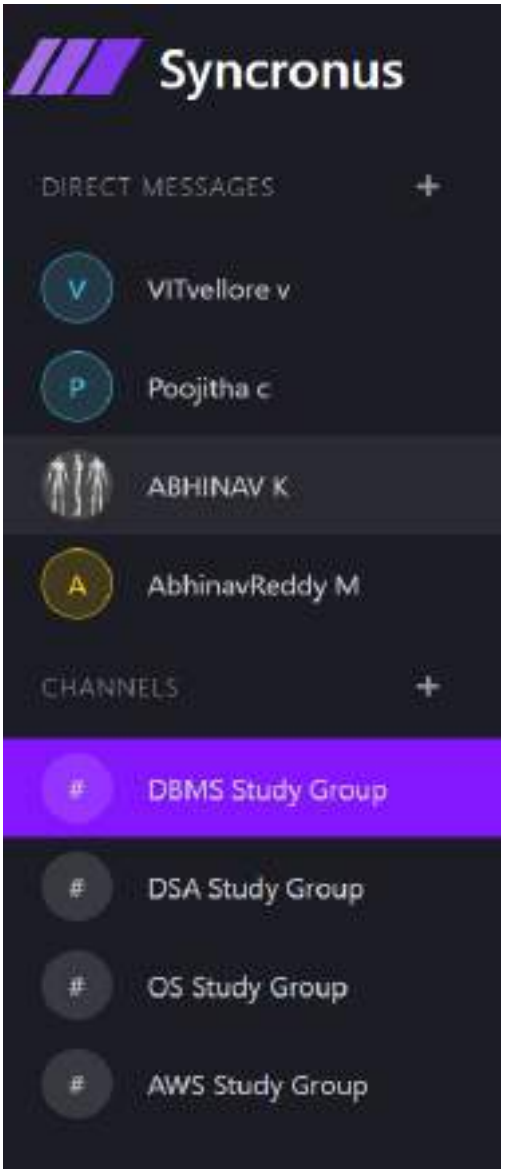


- **Read**: Retrieving chat history, viewing user profiles, listing chat rooms, chat rooms history.

JS Code to retrieve chat history of the given users:

```
export const getMessages = async (req, res, next) => {
  try {
    const user1 = req.userId;
    const user2 = req.body.id;
    if (!user1 || !user2) {
      return res.status(400).send("Both user IDs are required.");
    }

    const messages = await Message.find({
      $or: [
        { sender: user1, recipient: user2 },
        { sender: user2, recipient: user1 },
      ],
    }).sort({ timestamp: 1 });

    return res.status(200).json({ messages });
  } catch (err) {
    console.log(err);
    return res.status(500).send("Internal Server Error");
  }
};
```

Selecting a chat to view all of its chat history:



MongoDB chat retrieving with the help of Object ID:

JS Code for displaying user info:

```
export const getUserInfo = async (request, response, next) => {
  try {
    if (request.userId) {
      const userData = await User.findById(request.userId);
      if (userData) {
        return response.status(200).json({
          id: userData?.id,
          email: userData?.email,
          firstName: userData.firstName,
          lastName: userData.lastName,
          image: userData.image,
          profileSetup: userData.profileSetup,
          color: userData.color,
        });
      } else {
        return response.status(404).send("User with the given id not found.");
      }
    } else {
      return response.status(404).send("User id not found.");
    }
  } catch (error) {
    console.log({ error });
    return response.status(500).send("Internal Server Error");
  }
};
```

Retrieving user data in mongoDB:

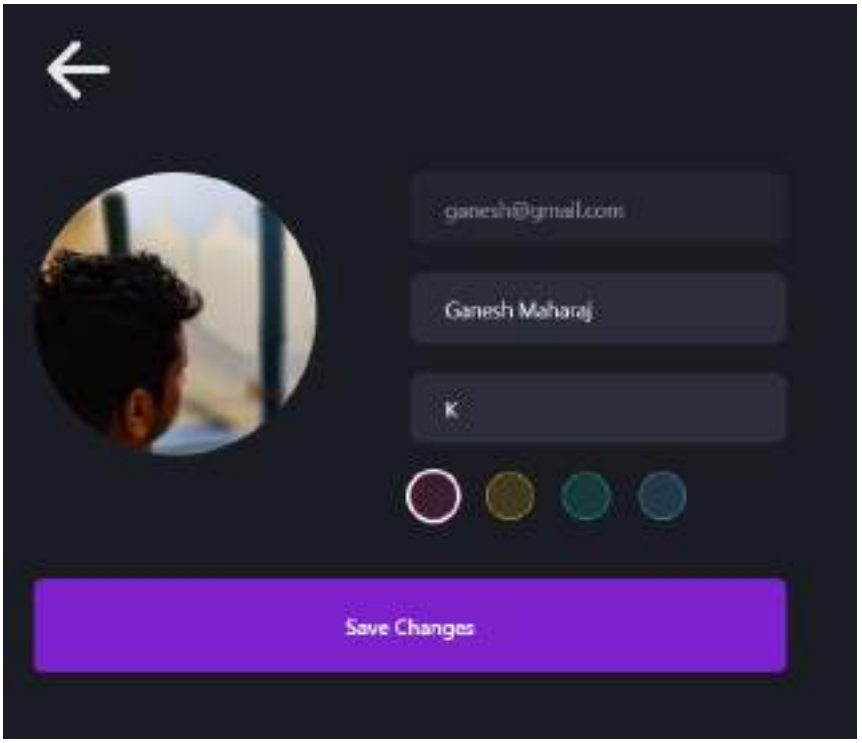
```
{_id:ObjectId('672ce8f5a389c7acaf27f491')}
```

Generate query | Explain

ADD DATA | EXPORT DATA | UPDATE | DELETE | 25

```
_id: ObjectId('672ce8f5a389c7acaf27f491')
email : "abhinavkonguva@gmail.com"
password : "$2b$10$UjqHXoJLUK8Uugbj3r8Az.5EZ02eiHC7vtIU1M91kaT3eH78RRh7i"
profileSetup : true
__v : 0
color : 2
firstName : "ABHINAV"
lastName : "K"
image : "uploads/profiles/1731059693283 6e384f2cfa76c8cb08e9d7c4f52f8696.jpg"
```

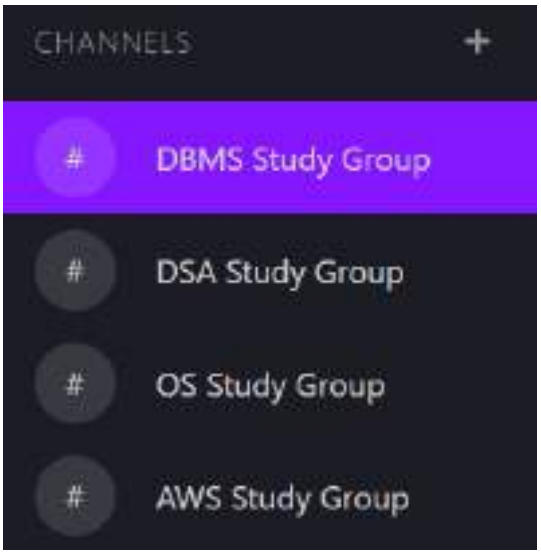
Displaying User Profile:



JS Code for showing all the channels of the user:

```
export const getUserChannels = async (req, res) => {
  try {
    const userId = new mongoose.Types.ObjectId(req.userId);
    const channels = await Channel.find({
      $or: [{ admin: userId }, { members: userId }],
    }).sort({ updatedAt: -1 });

    return res.status(200).json({ channels });
  } catch (error) {
    console.error("Error getting user channels:", error);
    return res.status(500).json({ message: "Internal Server Error" });
  }
};
```

Listing the channels of the user:



List of Users(indicated by their unique ID) in a chat room using MongoDB:

```
_id: ObjectId('672ceb44a389c7acaf27f508')
name : "DBMS Study Group"
▾ members : Array (3)
    0: ObjectId('672ce96fa389c7acaf27f49a')
    1: ObjectId('672ce8f5a389c7acaf27f491')
    2: ObjectId('672ce8a8a389c7acaf27f484')
admin : ObjectId('672ce772a389c7acaf27f470')
▸ messages : Array (19)
createdAt : 2024-11-07T16:31:00.319+00:00
updatedAt : 2024-11-07T16:48:39.431+00:00
__v : 0
```

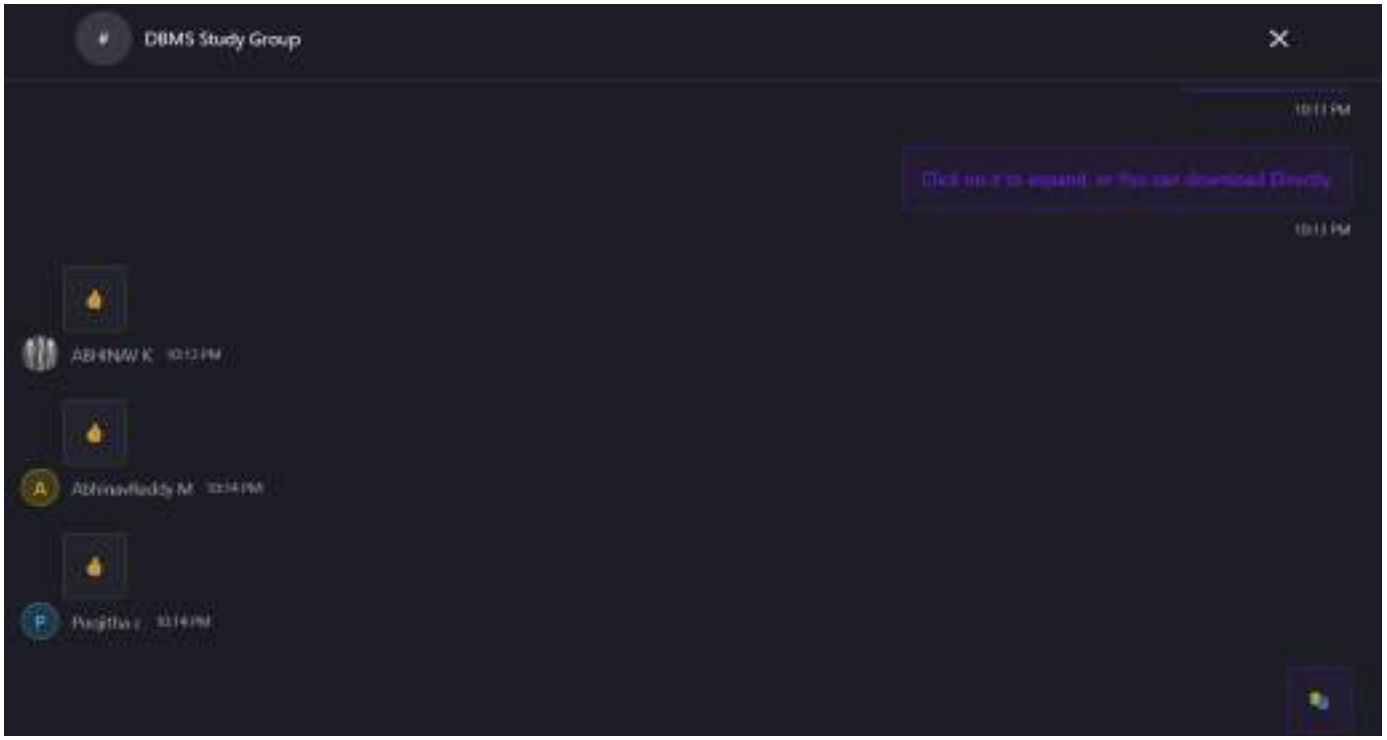
JS Code for retrieving the messages in a channel:

```javascript
export const getChannelMessages = async (req, res, next) => {
  try {
    const { channelId } = req.params;

    const channel = await Channel.findById(channelId).populate({
      path: "messages",
      populate: {
        path: "sender",
        select: "firstName lastName email _id image color",
      },
    });

    if (!channel) {
      return res.status(404).json({ message: "Channel not found" });
    }

    const messages = channel.messages;
    return res.status(200).json({ messages });
  } catch (error) {
    console.error("Error getting channel messages:", error);
    return res.status(500).json({ message: "Internal Server Error" });
  }
};
```

List of all the messages sent in a chat room using MongoDB:

```
_id: ObjectId('672ceb44a389c7acaf27f508')
name : "DBMS Study Group"
▸ members : Array (3)
admin : ObjectId('672ce772a389c7acaf27f470')
▾ messages : Array (19)
    0: ObjectId('672ceb94a389c7acaf27f518')
    1: ObjectId('672ceb9ca389c7acaf27f51f')
    2: ObjectId('672cebbea389c7acaf27f526')
    3: ObjectId('672cebe4a389c7acaf27f52d')
    4: ObjectId('672cec40a389c7acaf27f53b')
    5: ObjectId('672cec4ea389c7acaf27f542')
    6: ObjectId('672cec72a389c7acaf27f54c')
    7: ObjectId('672ceca2a389c7acaf27f553')
    8: ObjectId('672cecaea389c7acaf27f55a')
    9: ObjectId('672cecd5a389c7acaf27f564')
    10: ObjectId('672ced03a389c7acaf27f56b')
    11: ObjectId('672ced94a389c7acaf27f572')
    12: ObjectId('672cedf2a389c7acaf27f579')
    13: ObjectId('672cee19a389c7acaf27f580')
    14: ObjectId('672cee2fa389c7acaf27f587')
    15: ObjectId('672cee4da389c7acaf27f58e')
    16: ObjectId('672cee5ea389c7acaf27f595')
    17: ObjectId('672cee73a389c7acaf27f59c')
    18: ObjectId('672cef67a389c7acaf27f5d4')
createdAt : 2024-11-07T16:31:00.319+00:00
updatedAt : 2024-11-07T16:48:39.431+00:00
__v : 0
```

Sneak peek on a chat room:

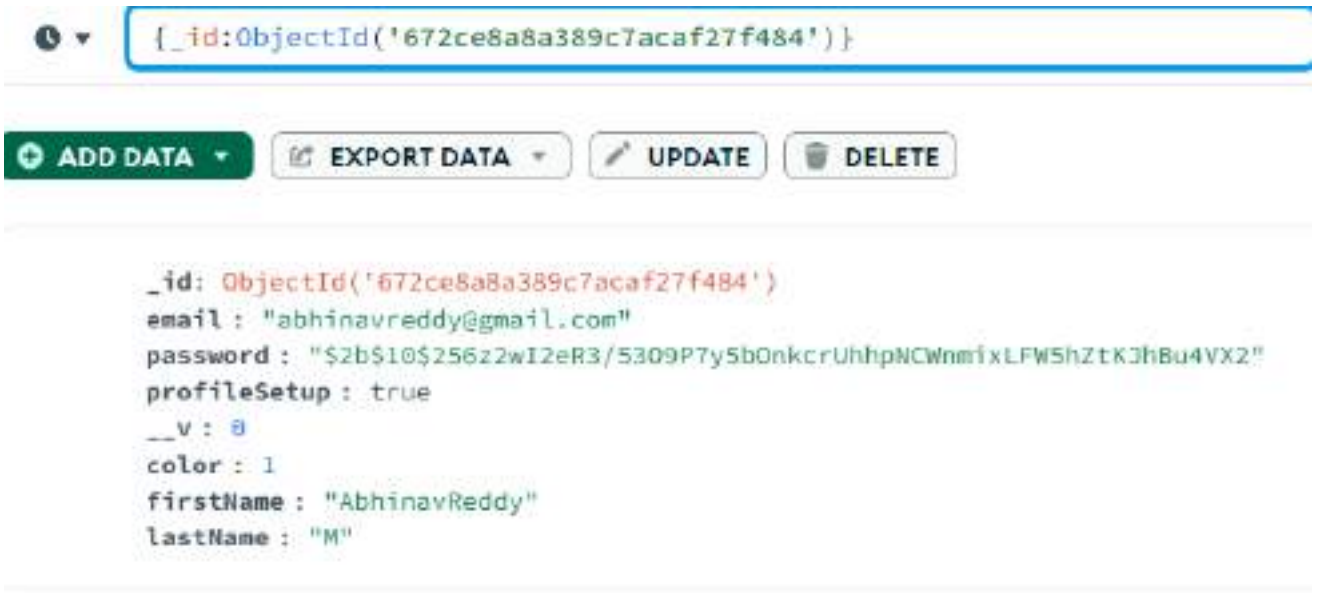


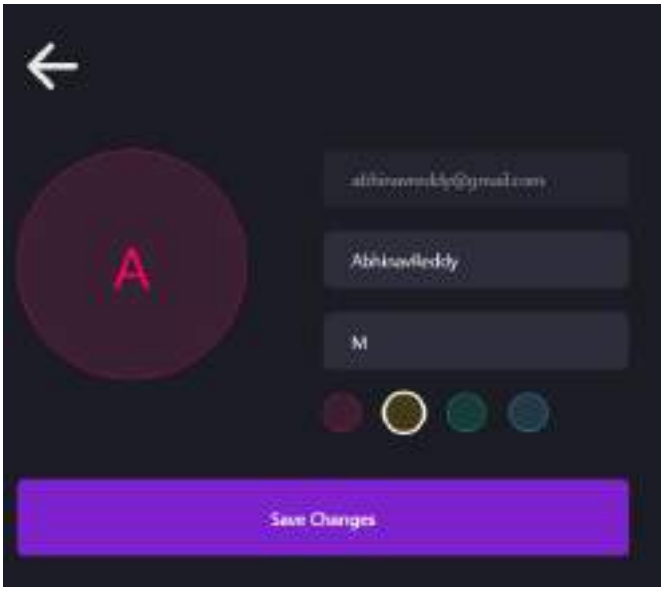
- **Update: Updating profile info.**

JS Code for updating profile info of user:

```
export const updateProfile = async (request, response, next) => {
  try {
    const { userId } = request;

    const { firstName, lastName, color } = request.body;

    if (!userId) {
      return response.status(400).send("User ID is required.");
    }

    if (!firstName || !lastName) {
      return response.status(400).send("Firstname and last name is required.");
    }

    const userData = await User.findByIdAndUpdate(
      userId,
      {
        firstName,
        lastName,
        color,
        profileSetup: true,
      },
      {
        new: true,
        runValidators: true,
      }
    );
    return response.status(200).json({
      id: userData.id,
      email: userData.email,
      firstName: userData.firstName,
      lastName: userData.lastName,
      image: userData.image,
      profileSetup: userData.profileSetup,
      color: userData.color,
    });
  } catch (error) {
    return response.status(500).send("Internal Server Error.");
  }
};
```

Before updating user profile:





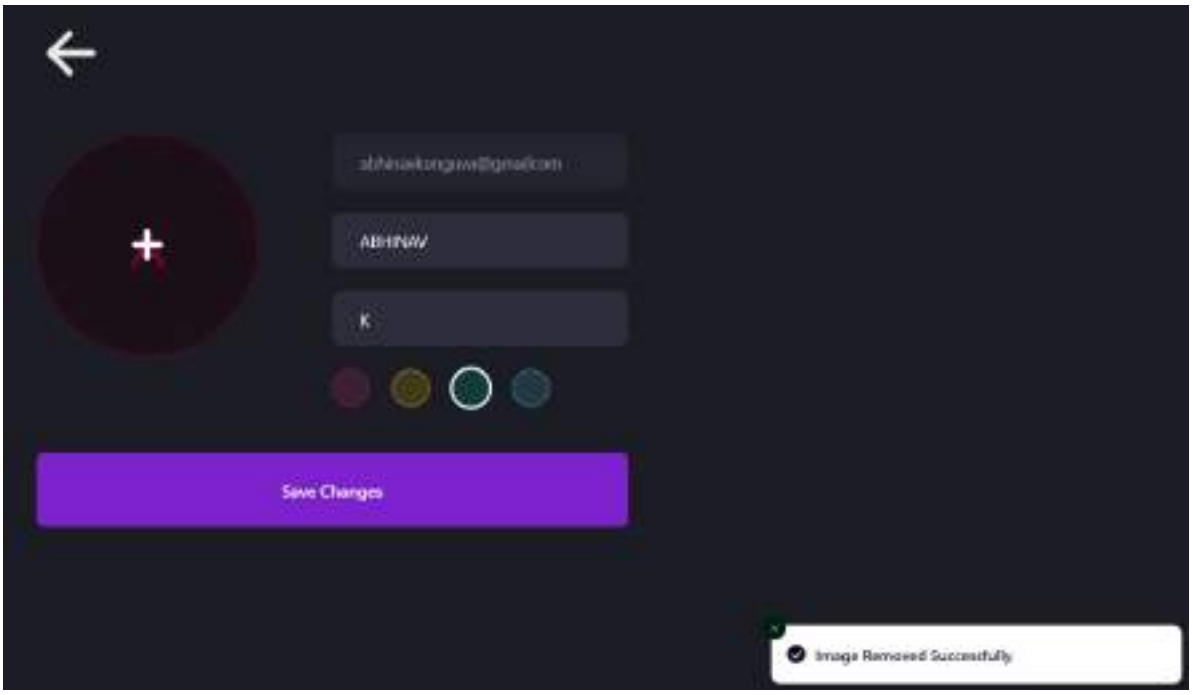
After updating user profile:

- **Delete**: Removing User picture, Delete User

JS Code for removing profile picture:

```
export const removeProfileImage = async (request, response, next) => {
  try {
    const { userId } = request;

    if (!userId) {
      return response.status(400).send("User ID is required.");
    }

    const user = await User.findById(userId);

    if (!user) {
      return response.status(404).send("User not found.");
    }

    if (user.image) {
      unlinkSync(user.image);
    }

    user.image = null;
    await user.save();

    return response
      .status(200)
      .json({ message: "Profile image removed successfully." });
  } catch (error) {
    console.log({ error });
    return response.status(500).send("Internal Server Error.");
  }
};
```

Removing profile picture:



```
_id: ObjectId('672ce8f5a389c7acaf27f491')
email : "abhinavkenguva@gmailcom"
password : "$2b$10$KTsbmx7Y33/zMK1X0yjG5u7VOa5Nq2P9L6S8I.AA82jCeM9LAqti6"
profileSetup : true
__v : 0
color : 2
firstName : "ABHINAV"
lastName : "K"
image : null
```

null

JS Code for deleting a user account:

```
const deleteUser = async (request, response) => {
  const { userId } = request.params;

  try {
    const deletedUser = await User.findByIdAndDelete(userId);

    if (!deletedUser) {
      return response.status(404).json({ message: "User not found." });
    }

    return response.status(200).json({
      message: "User deleted successfully.",
      id: deletedUser.id,
      email: deletedUser.email,
      firstName: deletedUser.firstName,
      lastName: deletedUser.lastName,
    });
  } catch (error) {
    return response.status(500).send("Internal Server Error.");
  }
};
```

Questions:

Q1: What methods did you implement to add and retrieve messages between users?

- **Answer**: We used POST requests to add messages to the Messages collection, including senderID, chatRoomID, content, and timestamp. For retrieval, GET requests query messages by chatRoomID, sorted by timestamp for ordered history. This setup allows users to post and view messages in real time.

Q2: How did you handle updates and deletions to maintain data consistency in your application?

- **Answer**: For updates, only the message sender or admin can modify content, using PUT requests to update specific fields like content. For deletions, DELETE requests remove messages but leave a placeholder (e.g., “message deleted”) to maintain continuity. This approach preserves the chat history flow.

3. Indexing

Explanation: Indexing enhances query performance by making data retrieval faster. For this application:

- Indexes on message timestamps speed up searching for messages within specific time ranges.
- Indexes on chat room IDs allow quick access to chat room-related data, especially for listing active chat rooms.

Indexing the messages according to timestamp in MongoDB:

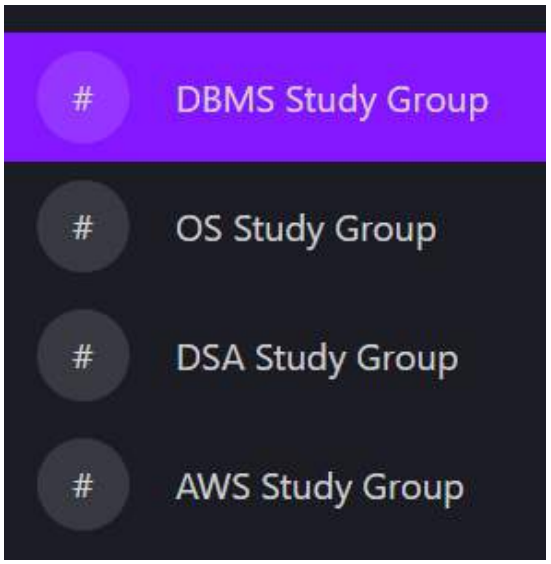
```
▸ _id: ObjectId('672ceb44a389c7acaf27f508')
  name : "DBMS Study Group"
▸ members : Array (3)
  admin : ObjectId('672ce772a389c7acaf27f470')
▾ messages : Array (19)
    0: ObjectId('672ceb94a389c7acaf27f518')
    1: ObjectId('672ceb9ca389c7acaf27f51f')
    2: ObjectId('672cebbea389c7acaf27f526')
    3: ObjectId('672cebe4a389c7acaf27f52d')
    4: ObjectId('672cec40a389c7acaf27f53b')
    5: ObjectId('672cec4ea389c7acaf27f542')
    6: ObjectId('672cec72a389c7acaf27f54c')
    7: ObjectId('672ceca2a389c7acaf27f553')
    8: ObjectId('672cecaea389c7acaf27f55a')
    9: ObjectId('672cecd5a389c7acaf27f564')
    10: ObjectId('672ced03a389c7acaf27f56b')
    11: ObjectId('672ced94a389c7acaf27f572')
    12: ObjectId('672cedf2a389c7acaf27f579')
    13: ObjectId('672cee19a389c7acaf27f580')
    14: ObjectId('672cee2fa389c7acaf27f587')
    15: ObjectId('672cee4da389c7acaf27f58e')
    16: ObjectId('672cee5ea389c7acaf27f595')
    17: ObjectId('672cee73a389c7acaf27f59c')
    18: ObjectId('672cef67a389c7acaf27f5d4')
  createdAt : 2024-11-07T16:31:00.319+00:00
  updatedAt : 2024-11-07T16:48:39.431+00:00
  __v : 0
```

JS Code for formatting timestamp:

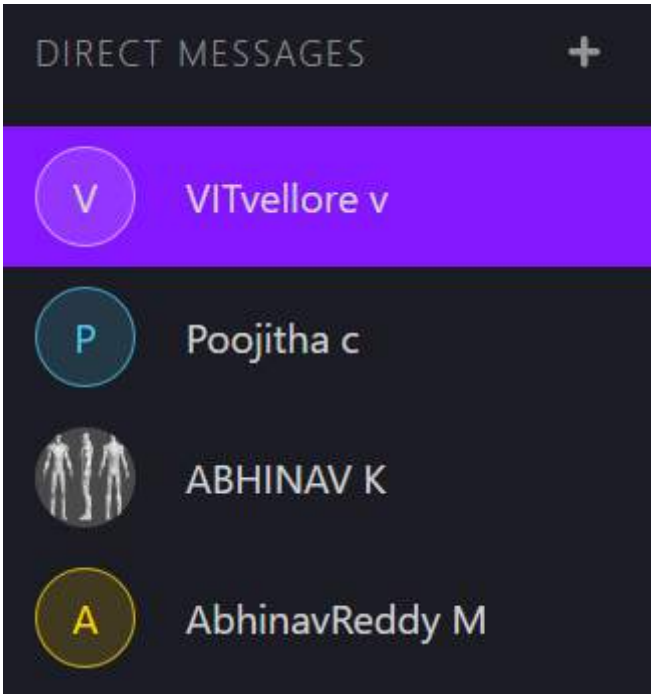
```
  messages: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "Messages",
      required: false,
    },
  ],
  createdAt: {
    type: Date,
    default: Date.now,
  },
  updatedAt: {
    type: Date,
    default: Date.now,
  },
});
```

Order of channels before sending a message in DBMS group:

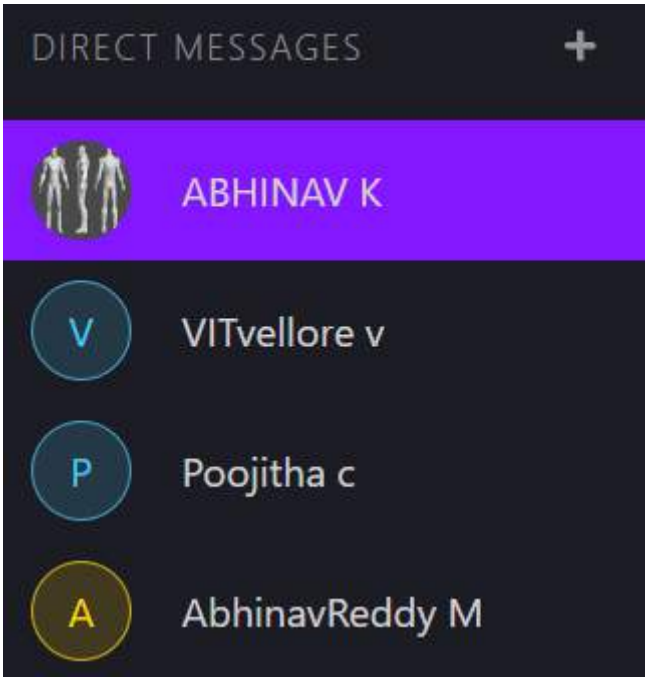
Order of channel after sending a message in DBMS group:



Order of Direct chats before sending a message to Abhinav K:



Order of Direct chats after sending a message to Abhinav K:



Questions:

Q1: How did you choose which fields to index in your database, and why?

- **Answer**: I indexed chatRoomID and timestamp in the Messages collection to speed up retrieval of chat history for specific rooms and to display messages chronologically. These indexes optimize real-time messaging performance, especially for active chat rooms.

Q2: What improvements in query performance did you observe after implementing indexes?

- **Answer**: Indexing significantly reduced response times for message retrieval, especially in busy chat rooms, by allowing MongoDB to quickly locate and retrieve relevant messages. This improved overall responsiveness for users accessing active conversations.

---

4. Aggregation

Explanation: Aggregation allows data processing across multiple records to provide insights. Examples:

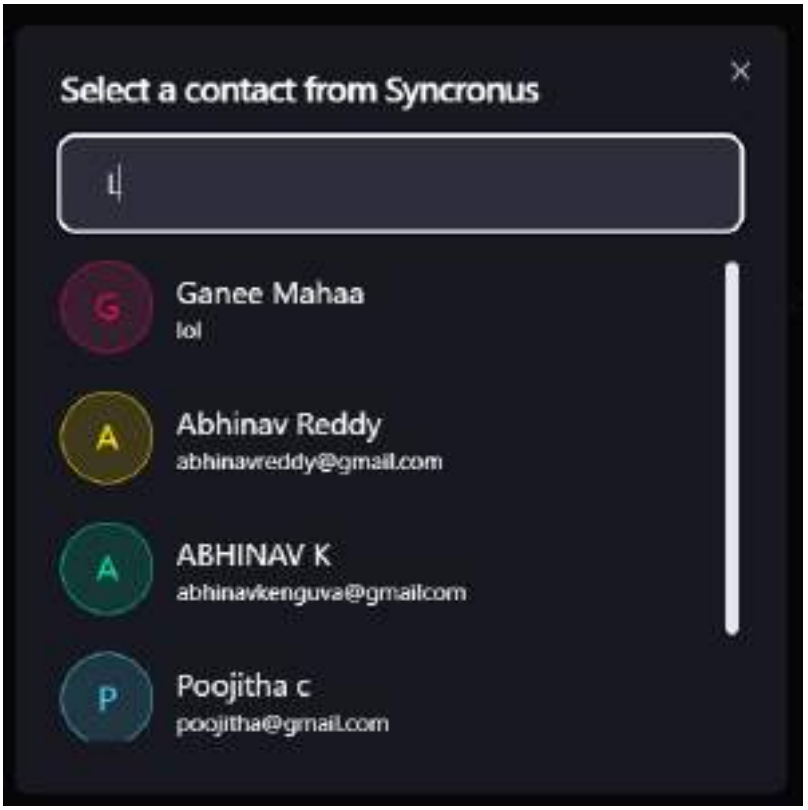
- Aggregating message data to identify the most interacted with the user or chat room.
- Counting the number of messages per chat room or per user.

JS Code for aggregating users based on search request:

```js
export const getContactsForList = async (req, res, next) => {
  try {
    let { userId } = req;
    userId = new mongoose.Types.ObjectId(userId);

    if (!userId) {
      return res.status(400).send("User ID is required.");
    }
    const contacts = await Message.aggregate([
      {
        $match: {
          $or: [{ sender: userId }, { recipient: userId }],
        },
      },
      {
        $sort: { timestamp: -1 },
      },
      {
        $group: {
          _id: {
            $cond: {
              if: { $eq: ["$sender", userId] },
              then: "$recipient",
              else: "$sender",
            },
          },
          lastMessageTime: { $first: "$timestamp" },
        },
      },
      {
        $lookup: {
          from: "users",
          localField: "_id",
          foreignField: "_id",
          as: "contactInfo",
        },
      },
      {
        $unwind: "$contactInfo",
      },
```

```
94          {
95            $project: {
96              _id: 1,
97
98              lastMessageTime: 1,
99              email: "$contactInfo.email",
100             firstName: "$contactInfo.firstName",
101             lastName: "$contactInfo.lastName",
102             image: "$contactInfo.image",
103             color: "$contactInfo.color",
104           },
105         },
106         {
107           $sort: { lastMessageTime: -1 },
108         },
109       ]);
110
111       return res.status(200).json({ contacts });
112     } catch (error) {
113       console.error("Error getting user contacts:", error);
114       return res.status(500).send("Internal Server Error");
115     }
116   };
117
```

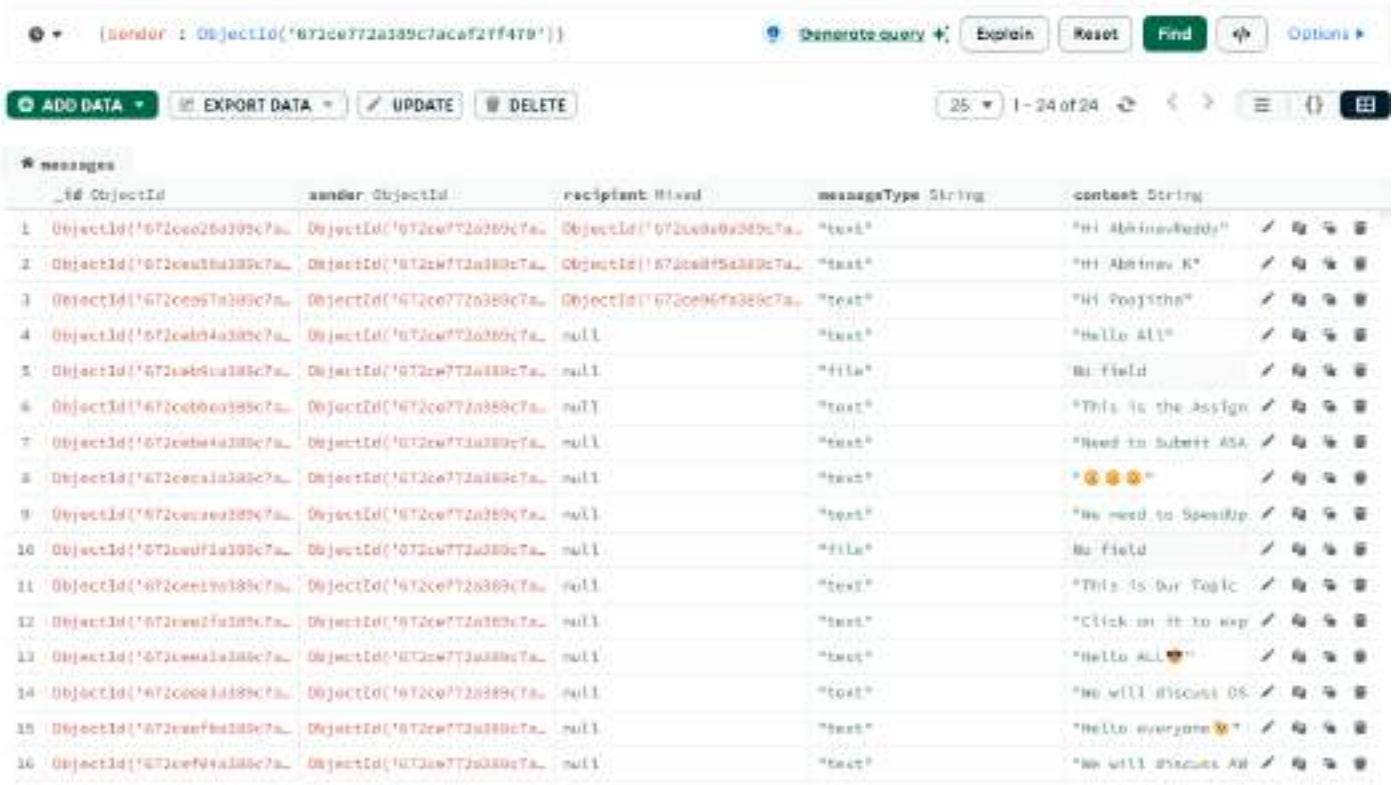
Aggregating user based on search value from most relevant to least relevant:



Aggregating data in MongoDB:

```
_id: ObjectId('66e4ff719da239f2e9ac18ce')
email : "lol"
password : "$2b$10$H3kuyvym7Bd8dA2U2yxyDeEA8g4…
profileSetup : true
__v : 0
color : 0
firstName : "Ganee"
lastName : "Mahaa"
```

List of all the messages in a chat room:



JS Code for obtaining the list of active chat rooms:

```
const getActiveChatRooms = async () => {
  return await Message.aggregate([
      { $group: { _id: "$chatRoomID", messageCount: { $sum: 1 } } },
      { $sort: { messageCount: -1 } }
  ]);
};
```

List of active chat rooms ordered from most active to least active:

Questions:

Q1: What types of aggregated data do you provide to show chat room activity?

- **Answer**: Aggregation pipelines track the number of messages per chat room and identify popular users based on message counts. I also aggregate the number of active chat rooms and display recent activity statistics to show user engagement.

Q2: How did you implement aggregation to find popular users or chat rooms?

- **Answer**: I used $group to aggregate message counts by senderID for popular users and by chatRoomID to identify active chat rooms. Sorting these results allows me to highlight the most active users and rooms in the application's analytics.

---

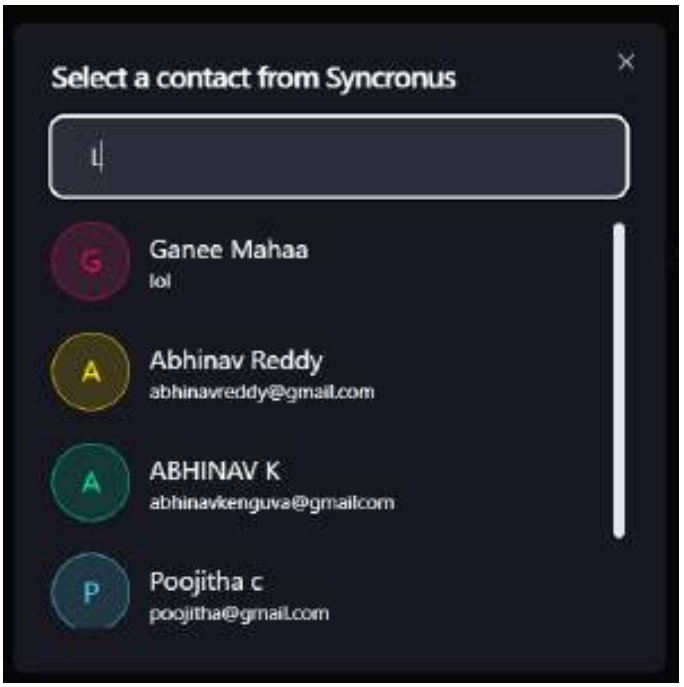
5. Text Search

Explanation: Full-text search lets users search through message content. MongoDB's text indexes allow searching words within message content, enabling keyword-based message retrieval.

JS Code for searching users:

```
export const searchContacts = async (request, response, next) => {
  try {
    const { searchTerm } = request.body;

    if (searchTerm === undefined || searchTerm === null) {
      return response.status(400).send("Search Term is required.");
    }

    const sanitizedSearchTerm = searchTerm.replace(
      /[.*+?^${}()|[\]\\]/g,
      "\\$&"
    );

    const regex = new RegExp(sanitizedSearchTerm, "i");

    const contacts = await User.find({
      $and: [
        { _id: { $ne: request.userId } },
        {
          $or: [{ firstName: regex }, { lastName: regex }, { email: regex }],
        },
      ],
    });
    return response.status(200).json({ contacts });
  } catch (error) {
    console.log({ error });
    return response.status(500).send("Internal Server Error.");
  }
};
```

Displaying search results ordered from most relevant to least relevant:



Searching a user in MongoDB:

```
_id: ObjectId('66e4ff719da239f2e9ac18ce')
email : "lol"
password : "$2b$10$H3kuyvym7Bd8dA2U2yxyDeEA8g4…
profileSetup : true
__v : 0
color : 0
firstName : "Ganee"
lastName : "Mahaa"
```

Questions:

Q1: How did you set up full-text search for searching through message content?

- **Answer**: I created a text index on the content field in the Messages collection, allowing full-text search within the chat application. This feature lets users search for specific keywords across chat messages and quickly locate information.

Q2: How does your text search handle cases like partial matches or common keywords?

- **Answer**: MongoDB's text search supports exact matches by default, so I used regex for partial matches when required. Common keywords or stop words are excluded from searches to ensure relevant results, improving the user search experience.

---

6. Data Validation

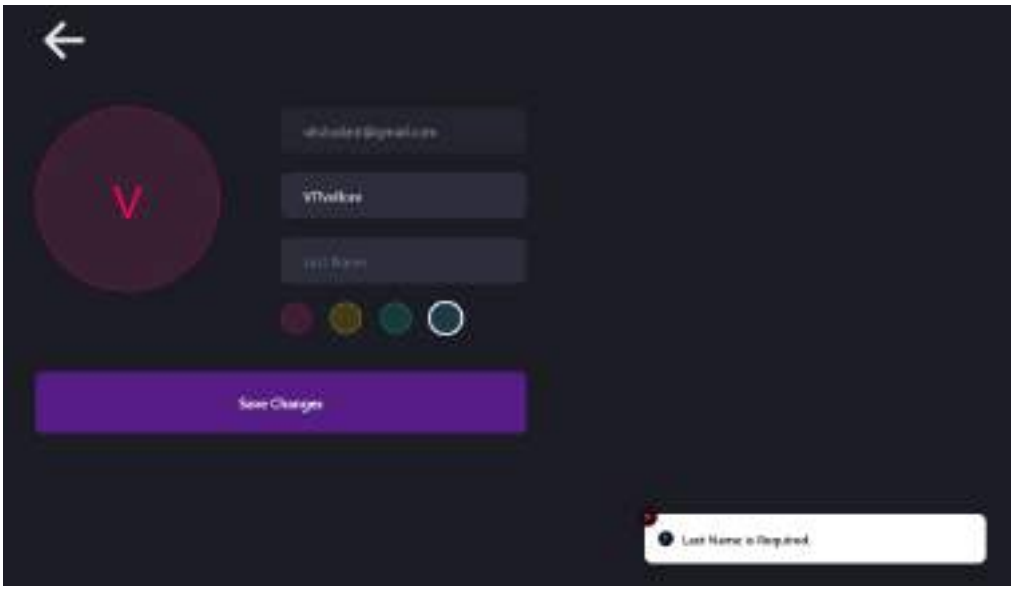
Explanation : Data validation ensures that only properly formatted data enters the database. This includes:

- Validating message content (e.g., non-empty, appropriate length).
- Ensuring user profiles contain all required fields in the correct format.

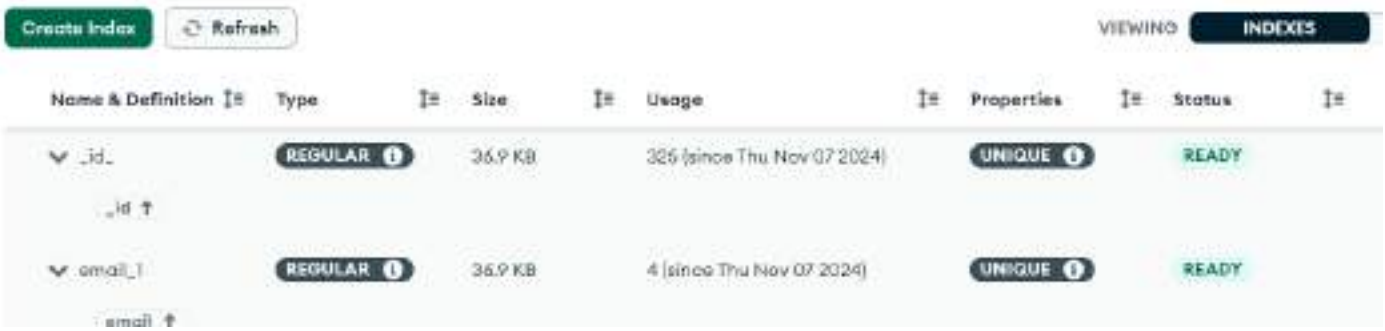
JS Code for validating data during signup:

```
export const signup = async (req, res, next) => {
  try {
    const { email, password } = req.body;
    if (email && password) {
      const user = await User.create({ email, password });
      res.cookie("jwt", createToken(email, user.id), {
        maxAge
      });

      return res.status(201).json({
        user: {
          id: user?.id,
          email: user?.email,
          firstName: user.firstName,
          lastName: user.lastName,
          image: user.image,
          profileSetup: user.profileSetup,
        },
      });
    } else {
      return res.status(400).send("Email and Password Required");
    }
  } catch (err) {
    console.log(err);
    return res.status(500).send("Internal Server Error");
  }
};
```

Rejecting invalid data during signup:



Storing the valid user data:



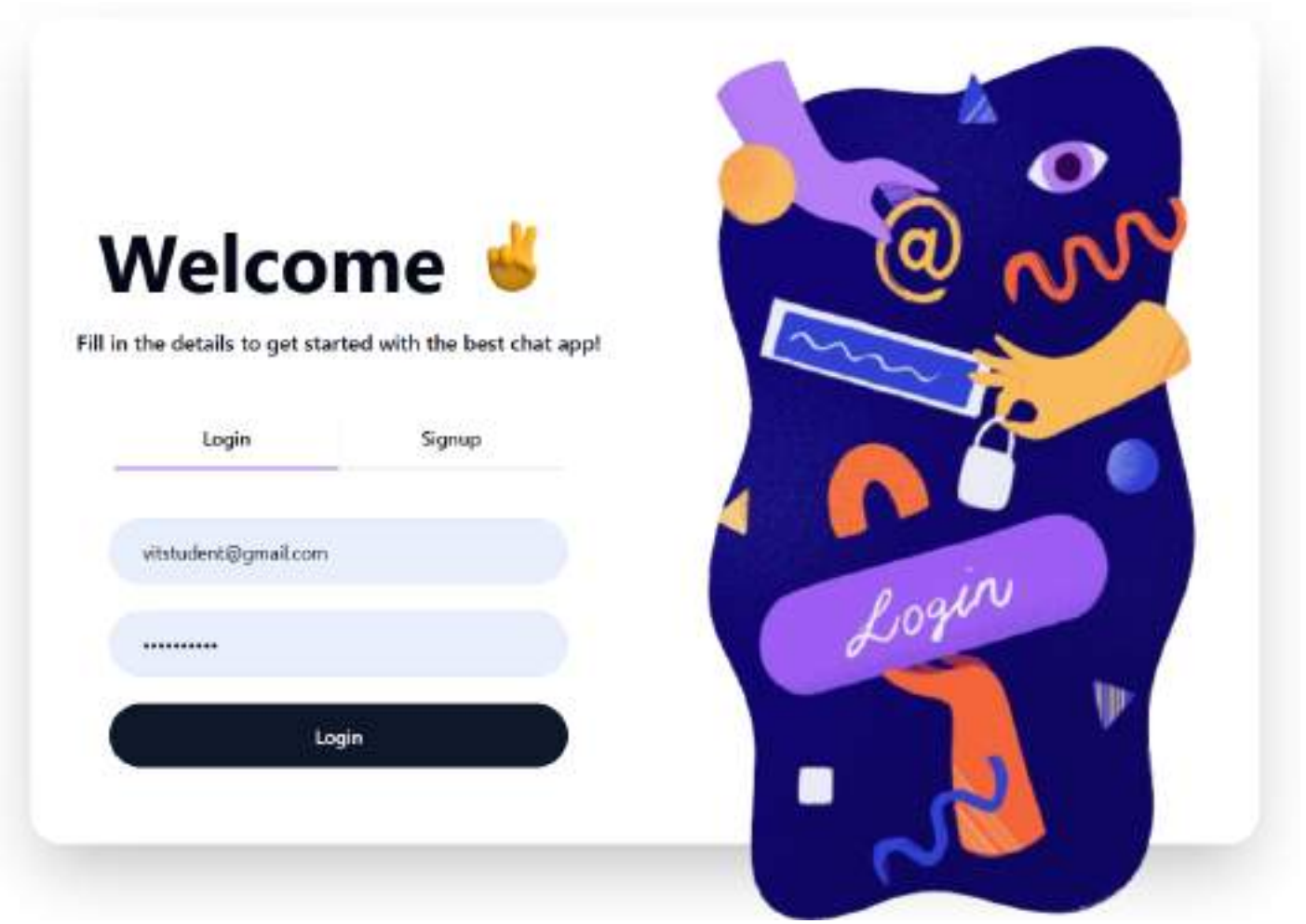
Create Index | Refresh | VIEWING INDEXES

Name & Definition	Type	Size	Usage	Properties	Status
_id_	REGULAR	36.9 KB	326 (since Thu Nov 07 2024)	UNIQUE	READY
_id ↑					
email_1	REGULAR	36.9 KB	4 (since Thu Nov 07 2024)	UNIQUE	READY
email ↑					

JS Code for validating data during login:

```
export const login = async (req, res, next) => {
  try {
    const { email, password } = req.body;
    if (email && password) {
      const user = await User.findOne({ email });
      if (!user) {
        return res.status(404).send("User not found");
      }
      const auth = await compare(password, user.password);
      if (!auth) {
        return res.status(400).send("Invalid Password");
      }
      res.cookie("jwt", createToken(email, user.id), {
        maxAge
      });
      return res.status(200).json({
        user: {
          id: user?.id,
          email: user?.email,
          firstName: user.firstName,
          lastName: user.lastName,
          image: user.image,
          profileSetup: user.profileSetup,
        },
      });
    } else {
      return res.status(400).send("Email and Password Required");
    }
  } catch (err) {
    return res.status(500).send("Internal Server Error");
  }
};
```

Login page in Website:



Questions:

Q1: What validation rules did you establish to ensure accurate data for messages and user profiles?

- **Answer**: Messages require non-empty content and are limited to a character count for better readability. User data, like username, email, and password, is validated with regex for formats (e.g., email format). Passwords are hashed before saving for security.

Q2: How do you handle cases when incoming data does not meet the validation criteria?

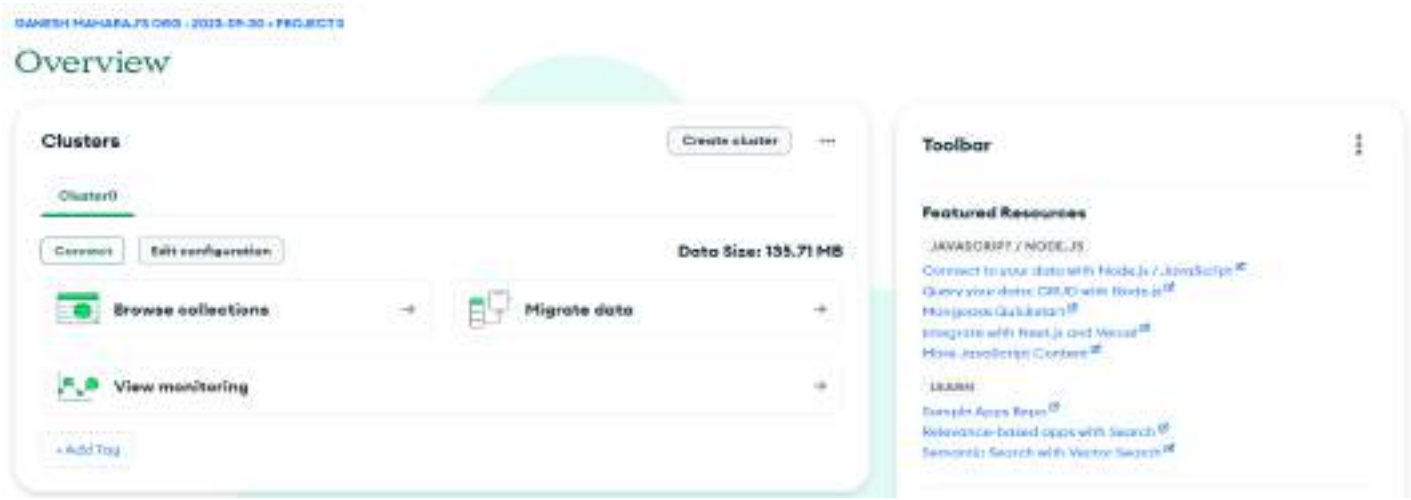
- **Answer**: When data doesn't meet validation rules, the API responds with descriptive error messages, prompting users to correct their input. This is handled in both the frontend with React forms and backend in Express, ensuring data quality.

---

7. Backup and Restore

Explanation: Regular backups protect data integrity, and restore operations allow data recovery in case of failure. MongoDB provides tools like mongodump and mongorestore for these operations**.**

Storing the data as backup in Atlas Cloud:



Questions :

Q1: What strategy did you adopt for performing regular backups of the chat data?

- **Answer**: We use MongoDb Atlas Cloud,

  Reliability: Atlas offers built-in redundancy and backups, which means your data is safer if the server has issues.

  Scalability: With Atlas, scaling your database becomes easier if your application grows.

  Security: Atlas provides more secure access with managed IP whitelisting and encrypted connections, helping you avoid potential exposure if Hostinger's security is limited.

  Global Accessibility: With Atlas, your app can access the database globally with minimal latency, which can improve performance for remote users.

Q2: How would you restore the data if a user accidentally deleted an important chat room?

- **Answer**: We use MongoDB Atlas as it provides a few mechanisms to help you recover data if it's accidentally deleted **:**

Continuous Backups: Atlas offers point-in-time recovery (PITR) for M10 clusters and above. This feature allows you to restore your data to a specific moment in time, which is especially useful if accidental deletions occur. You can roll back to any point in the past 24 hours with PITR.

Snapshot Backups: Atlas also performs regular, scheduled backups (every 6 hours for higher tiers) and stores them in a safe location. You can restore data from these snapshots if needed, though you might lose any changes made after the last snapshot.

Restore Tools: Atlas provides a self-service restore feature where you can create a new cluster or replace an existing one using a backup snapshot or point-in-time restore. This gives you a copy of the database state from before the data was deleted.

Automated Alerts and Monitoring: Atlas has monitoring and alerting features that can notify you when certain actions (like large deletions) happen, allowing you to take quick action.

---

8. Sharding

Explanation: Sharding distributes data across multiple servers, which helps manage high traffic or large data volumes. This application can shard by chat room IDs to distribute messages across shards based on chat rooms.

JS Code for setting up Sharding:

```javascript
const { MongoClient } = require('mongodb');

async function setupSharding() {
  try {
    // Connect to MongoDB router instance (mongos)
    const client = await MongoClient.connect("mongodb://localhost:27017", { useUnifiedTopology: true });
    const adminDb = client.db("admin");

    // Enable sharding on the database (replace "chatDB" with your database name if different)
    await adminDb.command({ enableSharding: "syncronus" });
    console.log("Sharding enabled on database syncronus.");

    // Shard the Messages collection on chatRoomId (hashed for even distribution)
    await adminDb.command({
      shardCollection: "syncronus.Messages",
      key: { chatRoomId: "hashed" }
    });
    console.log("Sharding configured on Messages collection by chatRoomId.");

    // Close the client connection
    client.close();
    console.log("Sharding setup complete.");
  } catch (err) {
    console.error("Error setting up sharding:", err);
  }
}

// Run the setup function
setupSharding();
```

Sharding in MongoDB:

```
_id: ObjectId('672ceb94a389c7acaf27f518')
sender : ObjectId('672ce772a389c7acaf27f470')
recipient : null
messageType : "text"
content : "Hello All"
timestamp : 2024-11-07T16:32:20.435+00:00
__v : 0
```

672ceb94a389c7acaf27f518

```
_id: ObjectId('672ceb9ca389c7acaf27f51f')
sender : ObjectId('672ce772a389c7acaf27f470')
recipient : null
messageType : "file"
fileUrl : "uploads/files/1730997148565/DBMS Assignment List A2 (1).pdf"
timestamp : 2024-11-07T16:32:28.578+00:00
__v : 0
```

Questions:

Q1: What benefits did sharding bring to your chat application in terms of scalability?

- **Answer**: Sharding distributes data across multiple servers, which improves scalability as the application grows. This setup allows high traffic handling without sacrificing query performance, making it suitable for a real-time chat application.

Q2: How did you set up sharding to handle a large number of messages efficiently?

- **Answer**: We used chatRoomID as the shard key to distribute messages based on the chat room. This reduces load on individual servers and ensures that messages are evenly distributed across multiple shards, enhancing scalability**.**

---

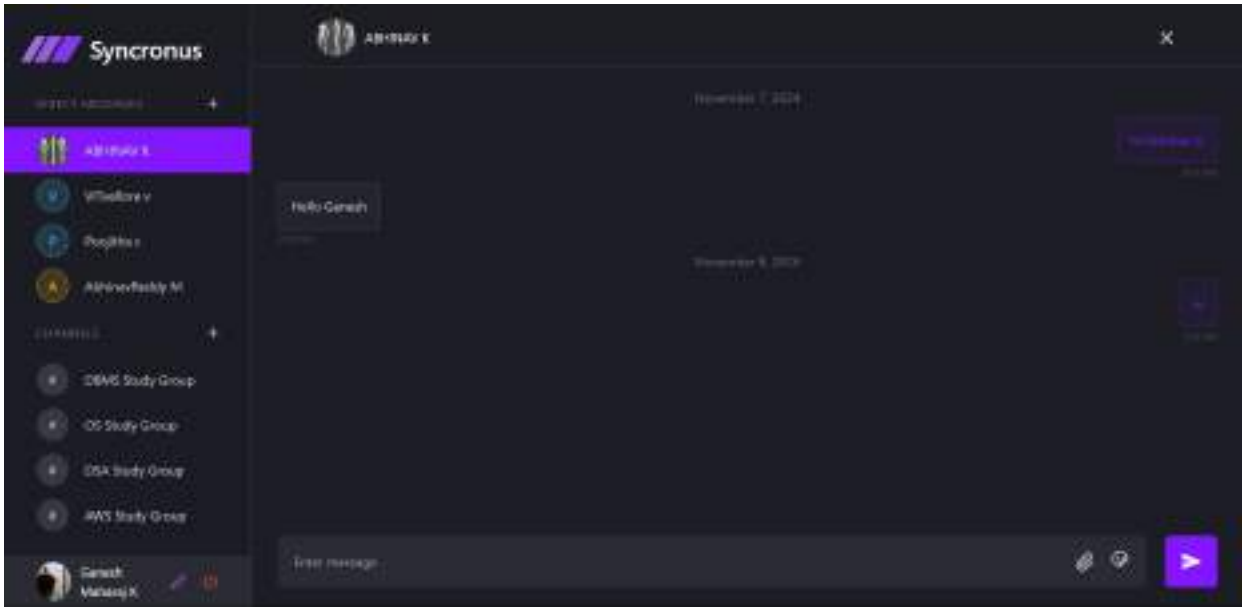
9. Change Streams

Explanation: Change streams allow real-time notifications of changes in MongoDB collections. For example, users can receive notifications for new messages in a chat room.

JS Code for changing streams:

```
const monitorMessages = async () => {
  const messageChangeStream = Message.watch();

  messageChangeStream.on('change', (change) => {
      if (change.operationType === 'insert') {
          const newMessage = change.fullDocument;
          console.log(`New message in chat room ${newMessage.chatRoomID}: ${newMessage.content}`);
      }
  });
};
monitorMessages();
```

Changing Streams flawlessly:



Questions:

Q1: How does your application use change streams to notify users of new messages?

- **Answer**: The backend listens to changes in the Messages collection using change streams. When a new message is added, it triggers a WebSocket event, notifying users in real-time. This approach allows instant message delivery.

Q2: How did you handle the performance impact of monitoring real-time updates across multiple chat rooms?

- **Answer**: To limit load, change streams are only active for chat rooms with online users. This selective approach optimizes performance by reducing the number of active streams and only monitoring the necessary collections.

---

10. Security

Explanation: Implementing security measures ensures that only authorized users can access chat rooms and messages. Authentication verifies user identity, and authorization controls access based on roles.

JS Code for user authentication during login:

```
export const login = async (req, res, next) => {
  try {
    const { email, password } = req.body;
    if (email && password) {
      const user = await User.findOne({ email });
      if (!user) {
        return res.status(404).send("User not found");
      }
      const auth = await compare(password, user.password);
      if (!auth) {
        return res.status(400).send("Invalid Password");
      }
      res.cookie("jwt", createToken(email, user.id), {
        maxAge
      });
      return res.status(200).json({
        user: {
          id: user?.id,
          email: user?.email,
          firstName: user.firstName,
          lastName: user.lastName,
          image: user.image,
          profileSetup: user.profileSetup,
        },
      });
    } else {
      return res.status(400).send("Email and Password Required");
    }
  } catch (err) {
    return res.status(500).send("Internal Server Error");
  }
};
```

List of authorized users in a chat room:

```
▸
  _id: ObjectId('672ceb44a389c7acaf27f508')
  name : "DBMS Study Group"
▾ members : Array (3)
    0: ObjectId('672ce96fa389c7acaf27f49a')
    1: ObjectId('672ce8f5a389c7acaf27f491')
    2: ObjectId('672ce8a8a389c7acaf27f484')
  admin : ObjectId('672ce772a389c7acaf27f470')
▸ messages : Array (24)
  createdAt : 2024-11-07T16:31:00.319+00:00
  updatedAt : 2024-11-08T10:22:02.013+00:00
  __v : 0
```

Questions:

Q1: What authentication and authorization mechanisms did you implement to secure chat rooms?

- **Answer**: The application uses JWT-based authentication, where users must log in to access chat rooms. Authorization checks ensure only users with valid JWT tokens can access specific chat rooms, protecting against unauthorized access.

Q2: How do you ensure that only authorized users can view or send messages in a chat room?

- **Answer**: Each chat room stores an array of authorized user IDs. When a user attempts to access a room, a backend check verifies the user's ID against this array. Unauthorized users are denied access, ensuring chat room security.
