



Aura ++

INSTAGRAM CLONE

Aura

Aura.png

first.txt

backend

node_modules

package-lock.json

package.json

frontend

Basic Files

Step1: npm init-y

We initialize the basic package.json file

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows files like `Aura.png`, `first.txt`, `package.json`, and `package-lock.json`.
- Editor:** A code editor window titled "first.txt" containing numbered steps for setting up a Node.js project.
- Terminal:** An integrated terminal window showing the command `npm init -y` being run in the directory `C:\Users\...INSTAGRAM CLONE\backend`. The output shows the creation of a `package.json` file with default values for keywords, author, license, and description.
- Output:** A panel showing the results of the `npm i` command, which added 129 packages and audited 130 packages in 13s. It also lists 21 packages looking for funding and found 0 vulnerabilities.
- Sidebar:** A sidebar on the right contains a tree view with three entries under "powershell".

Step2: npm i express mongoose dotenv cors bcryptjs
jsonwebtoken nodemon

- express -- is a web framework for node.js
- mongoose -- is a library for mongodb
- nodemon -- this will help us to restart the server automatically when we make changes in the code
- dotenv -- this will help us to manage environment variables
- jsonwebtoken -- this will help us to create and verify json web tokens
- bcryptjs -- this will help us to hash passwords
- cookie-parser -- this will help us to parse cookies
- cors -- this will help us to handle cross origin requests

also in the package.json -- we will add type:module
--using this we can use import and export statements
instead of require and module.exports

---"type": "module",

```
import express, { application } from 'express';
import cors from 'cors';
import connectDB from './utils/db.js';
// import cors from "cors";
```



Like here since we are using type module
So we have to write ./utils/db.js
Else we need to write ./utils/db

Running Nodemon

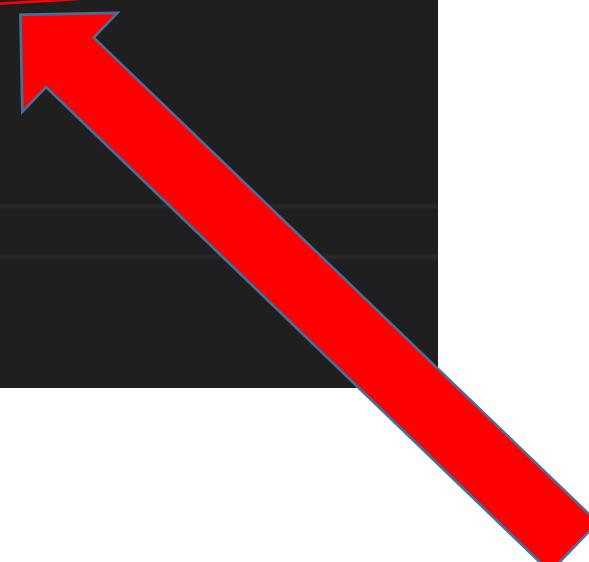
```
1  {
2    "name": "backend",
3    "version": "1.0.0",
4    "main": "index.js",
5    "type": "module",
6    "scripts": {
7      "dev": "nodemon index.js"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "description": "",
13   "dependencies": {
14     "bcryptjs": "^3.0.2",
15     "cookie-parser": "^1.4.7",
16     "cors": "^2.8.5",
17     "express": "^5.1.0",
18     "jsonwebtoken": "^9.0.2",
19     "mongoose": "^8.18.2",
20     "nodemon": "^3.1.10"
21   }
22 }
```

WHENEVER WE
RUN
Npm run dev
Then our server
will start

backend > **JS** index.js > ...

```
1 import express from 'express';
2 // import cors from "cors";
3 // import cookieParser from 'cookie-parser';
4
5 //till now we have intilize the project basic index.js file and run the server
6 const app = express();
7 const PORT = 5000;
8 app.listen(PORT, ()=>{
9   console.log(`server is running on port ${PORT}`);
10 })
```

index.js



BASIC DONE

Cookie parser and url encoded

```
app.use(cookieParser());
//whenever we try to request the token from browser..then we will store the token in the cookie parser

app.use(urlencoded({extended:true}));
//basically this will allow us to parse the incoming requests with urlencoded payloads
//this app.use(urlenconded) --
//this thing will allow the read of data which are sended in browser through post request
//extended:true -- this will allow us to parse nested objects
```

CORS(cross origin resource sharing)

```
//CORS - cross origin resource sharing
const corsOptions = {
  origin: "http://localhost:5173", // this 5173 is the port where our frontend is running(vite)
  credentials: true, // cookie send hogi agr hm credentails true krdenge
  //credentials: true allows cookies and authentication headers
  // to be sent and received between frontend and backend.
};

app.use(cors(corsOptions));
//this will allow the cross origin resource sharing
//frontend and backend can communicate with each other
```

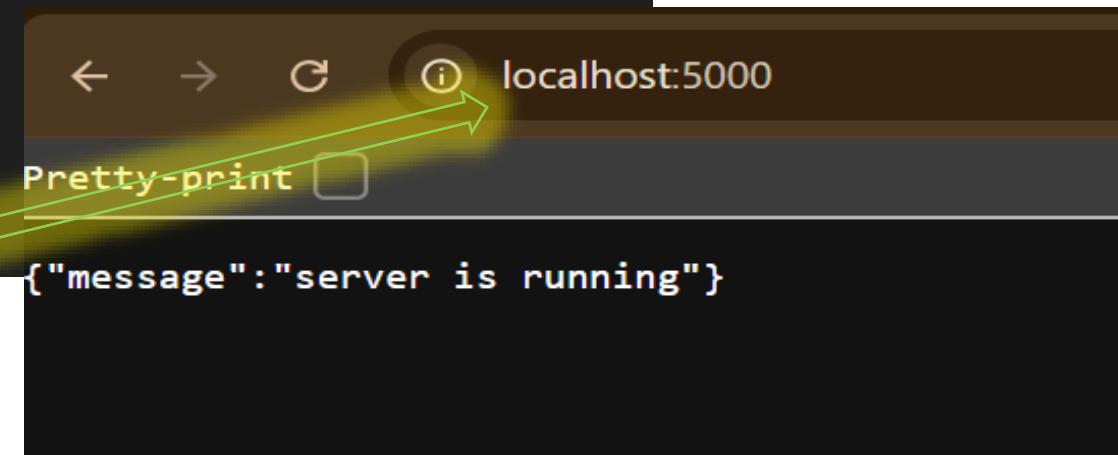
Credentials: true hoge
tabhi hm cookie ko
send kar payege

by default, browsers restrict cross-origin requests for security reasons. This means your frontend (on port 5173) cannot make requests to your backend (on port 5000) unless CORS is enabled. By using app.use(cors(corsOptions)), you explicitly allow your frontend to communicate with your backend, including sending cookies and credentials.

First API

```
app.get("/",(req,res)=>{
    return res.status(200).json([
        message:"server is running"
    ])
});
```

```
Node.js v22.16.0
[nodemon] app crashed - waiting for file changes before starting...
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
server is running on port 5000
```



Mongodb database working

A screenshot of the MongoDB Atlas dashboard. A large yellow arrow points down to the 'New Project' button, which is highlighted with a red circle. The dashboard shows a sidebar with various project names like 'sample', 'chat_app', 'Instagramclone', and 'Project 0'. The main area displays monitoring status for the 'sample' project, showing 'Monitoring for sample is paused.' and a link to visit documentation for more info. To the right, there's a 'Toolbar' section with tabs for 'Resources (9)', 'Tips (6)', and 'Performance' (which is selected). The 'Performance' tab includes sections for 'Performance Tools' (with a note about M10 clusters), 'Upgrades cluster', 'Create dedicated cluster', 'BACKUPS', and 'PERFORMANCE ADV!'. A chat interface at the bottom right shows a message from MBot asking how to get started.

← → ⌛ cloud.mongodb.com/v2/687092ca1907eb4b91cc3b25#/overview

Atlas arvind's Org ... Access Manager Billing All Clusters Get Help arvind

sample Services Charts

Search for a Project... -03-03 > SAMPLE

chat_app Instagramclone Project 0 sample

Create cluster ...

View All Projects + New Project

Data Federation

SECURITY

Quickstart

Backup

Database Access

Network Access

Advanced

New On Atlas 9 + Add Tag

Monitoring for sample is paused.

Monitoring will automatically resume when you connect to your cluster.

Visit the documentation for more info.

Toolbar

Resources (9) Tips (6) Performance

Performance Tools

Leverage tools for improved resilience and performance with an M10 cluster. Upgrade your Free/Flex cluster to unlock:

Upgrade cluster Create dedicated cluster

BACKUPS

Provide data protection and resilience with point-in-time recovery and geopoint.

PERFORMANCE ADV!

Hi, how can I help you get started? MBot • 3m ago

Receive index recommendations, write performance.

Need help creating my account?

REAL-TIME PERFORMANCE PANE

Identify critical operations; eval. Is Atlas right for me? Something else?

Create a Project

[Name Your Project](#)[Add Members](#)

Write name of project

Name Your Project

Project names have to be unique within the organization (and other restrictions).

Add Tags (Optional)

Use tags to efficiently label and categorize your projects. A project can have a maximum of 50 tags. You can modify tags for the project later. [Learn more](#)

Key	Value	Actions
application	: sparshsharma	
+ Add tag		1 TAG

[Cancel](#)[Next](#)

Create a Project

✓ Name Your Project

Add Members

Add Members and Set Permissions

Invite new or existing users via email address...

Give your members access permissions below.

81arvindarora@gmail.com
(you)

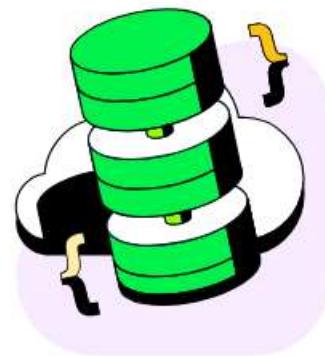
Project Owner

Back

Cancel

Create Project 

Overview



Create a cluster

Choose your cloud provider, region, and specs.

+ Create

Deploy your cluster

Use a template below or set up advanced configuration options. You can also edit these configuration options once the cluster is created.

M10

\$0.08/hour

Dedicated cluster for development environments and low-traffic applications.

STORAGE

10 GB

RAM

2 GB

vCPU

2 vCPUs

Flex

From \$0.011/hour

Up to \$30/month

For development and testing, with on-demand burst capacity for unpredictable traffic.

STORAGE

5 GB

RAM

Shared

vCPU

Shared

Free

For learning and exploring MongoDB in a cloud environment.

STORAGE

512 MB

RAM

Shared

vCPU

Shared

Configurations

Name

You cannot change the name once the cluster is created.

Cluster0

Quick setup

Automate security setup i

Preload sample dataset i

Provider



Region



★ Recommended i Low carbon emissions i

Tag (optional)

Create your first tag to categorize and label your resources; more tags can be added later. [Learn more.](#)

application

: sparshsharma



Hi, how can I help you get
MBot • 29m ago

Need help creatin

Is Atlas right for me?

Sc

I'll do this later

Go to Advanced Configuration

Create Deployment

Connect to Cluster0

1

Set up connection security

2

Choose a connection method

3

Connect



You need to secure your MongoDB Atlas cluster before you can use it. Set which users and IP addresses can access your cluster now. [Read more](#)

1. Add a connection IP address

Your current IP address (223.178.208.44) has been added to enable local connectivity. Only an IP address you add to your Access List will be able to connect to your project's clusters. Add more later in [Network Access](#).

2. Create a database user

This first user will have [atlasAdmin](#) permissions for this project.

We autogenerated a username and password. You can use this or create your own.

i You'll need your database user's credentials in the next step. Copy the database user password.

Username

81larvindarora_db_user

Password

hjjLjSkWiPoXqyDs

HIDE

Copy

Click to Copy Password

Create Database User

Close

Choose a connection method

Connect to Cluster0



Set up connection security



Choose a connection method



Connect

Connecting with MongoDB for VS Code

1. Install MongoDB for VS Code.

In [VS Code](#), open "Extensions" in the left navigation and search for "MongoDB for VS Code." Select the extension and click install.

2. In VS Code, open the Command Palette.

Click on "View" and open "Command Palette."

Search "MongoDB: Connect" on the Command Palette and click on "Connect with Connection String."

3. Connect to your MongoDB deployment.

Paste your connection string into the Command Palette.

Show Password [i](#)

`mongodb+srv://81arvindarora_db_user:hjjLjSkWiPoXqyDs@cluster0.udbefat.mongodb.net/`

The password for [81arvindarora_db_user](#) is included in the connection string for your first time setup. **This password will not be available again after exiting this connect flow.**

4. Click "Create New Playground" in MongoDB for VS Code to get started.

[Learn more about Playgrounds](#)

RESOURCES

[Connect to MongoDB through VSCode](#)

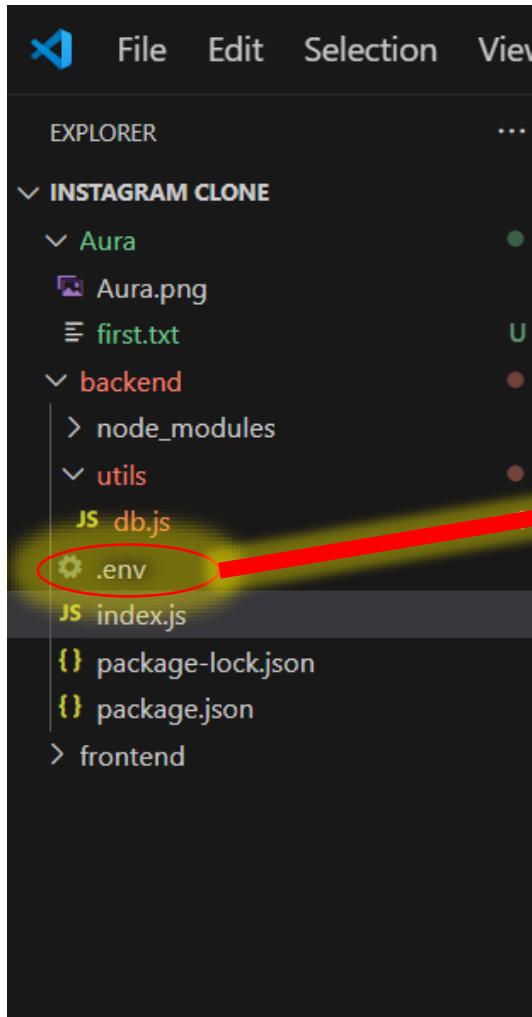
[Access your Database Users](#)

[Explore your data with playgrounds](#)

[Troubleshoot Connections](#)

We will write
This in
Environment
variables

Creating env file



Npm i dotenv to install it
Then in this folder we can write environment
Variables

```
File Edit Selection View ... ← → ⚡ INSTAGRAM CLONE  
EXPLORER ... Aura.png first.txt index.js db.js .env package.json  
INSTAGRAM CLONE backend > index.js > ...  
Aura Aura.png first.txt  
backend node_modules utils db.js .env index.js package-lock.json package.json  
frontend  
1 import cookieParser from 'cookie-parser';  
2 import express, { urlencoded } from 'express';  
3 import cors from 'cors';  
4 // import cors from "cors";  
5 // import cookieParser from 'cookie-parser';  
6  
7  
8 | import dotenv from 'dotenv';  
9  
10  
11 dotenv.config();  
12 //this will allow us to use the environment variables from the .env file  
13  
14 const app = express();  
15  
16  
17  
18
```

Here we have imported the dotenv file
It should be at the top
So that all variables should Load properly

It will load the environment variables

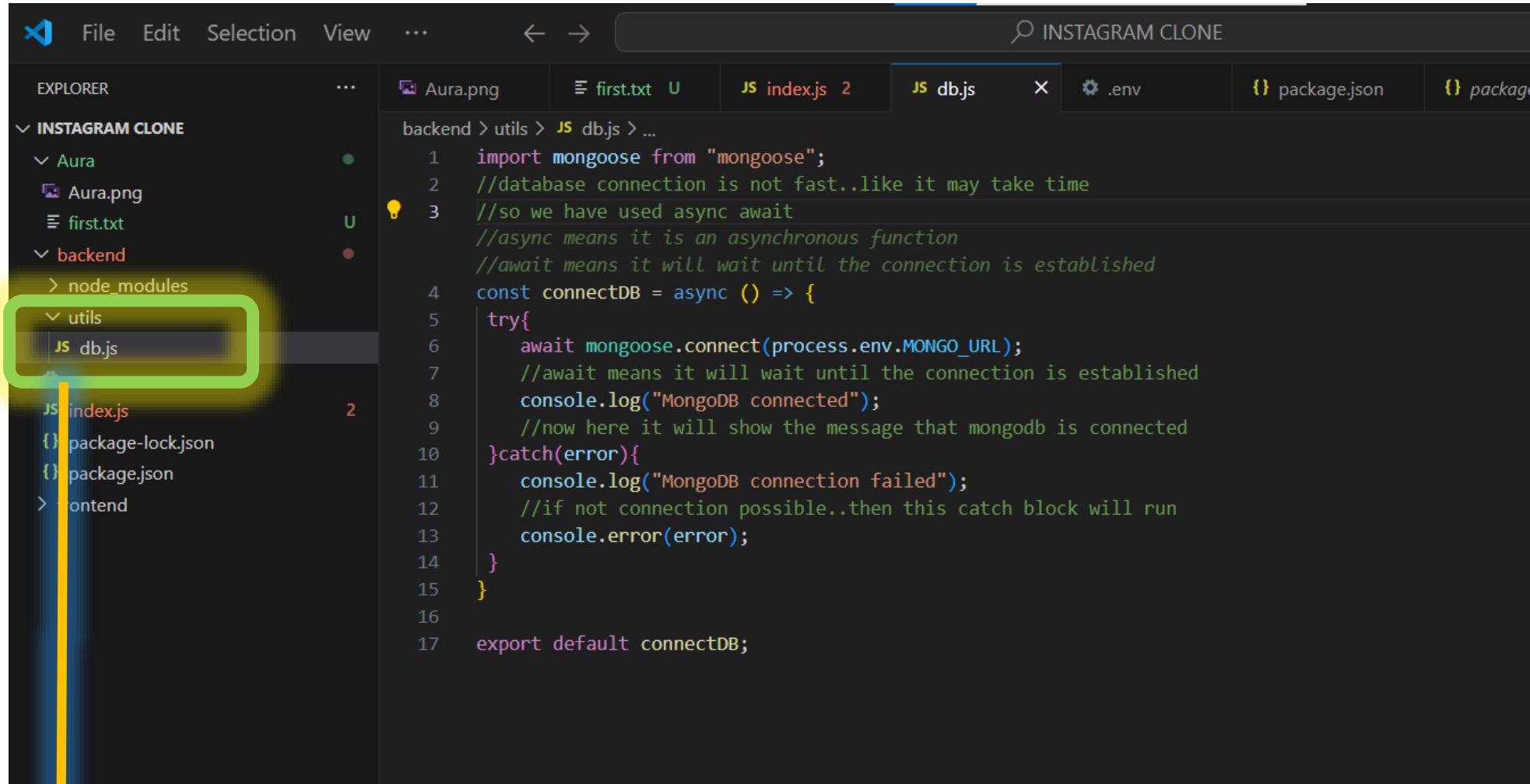
```
const PORT = process.env.PORT || 5000;
```

This is the way of accessing the variables of env file...

`process.env.variable_name`

here we have used `||` means if it is not available then use 5000

Connecting with database



The screenshot shows the VS Code interface with the title bar "INSTAGRAM CLONE". The Explorer sidebar on the left lists the project structure:

- INSTAGRAM CLONE
 - Aura
 - backend
 - node_modules
 - utils
 - db.js
 - index.js
 - package-lock.json
 - package.json
 - frontend

The "db.js" file in the "utils" folder is highlighted with a yellow box. The code editor shows the content of "db.js":

```
1 import mongoose from "mongoose";
2 //database connection is not fast..like it may take time
3 //so we have used async await
//async means it is an asynchronous function
//await means it will wait until the connection is established
4 const connectDB = async () => {
5   try{
6     await mongoose.connect(process.env.MONGO_URL);
7     //await means it will wait until the connection is established
8     console.log("MongoDB connected");
9     //now here it will show the message that mongodb is connected
10    }catch(error){
11      console.log("MongoDB connection failed");
12      //if not connection possible..then this catch block will run
13      console.error(error);
14    }
15  }
16
17 export default connectDB;
```

Creating db.js file inside utils folder

```
nd > utils > db.js > ...
import mongoose from "mongoose";
//database connection is not fast..like it may take time
//so we have used async await
//async means it is an asynchronous function
//await means it will wait until the connection is established
const connectDB = async () => {
  try{
    await mongoose.connect(process.env.MONGO_URL);
    //await means it will wait until the connection is established
    console.log("MongoDB connected");
    //now here it will show the message that mongodb is connected
  }catch(error){
    console.log("MongoDB connection failed");
    //if not connection possible..then this catch block will run
    console.error(error);
  }
}

export default connectDB;
```

Npm | mongoose

Database connection
Takes time..so using
async and await

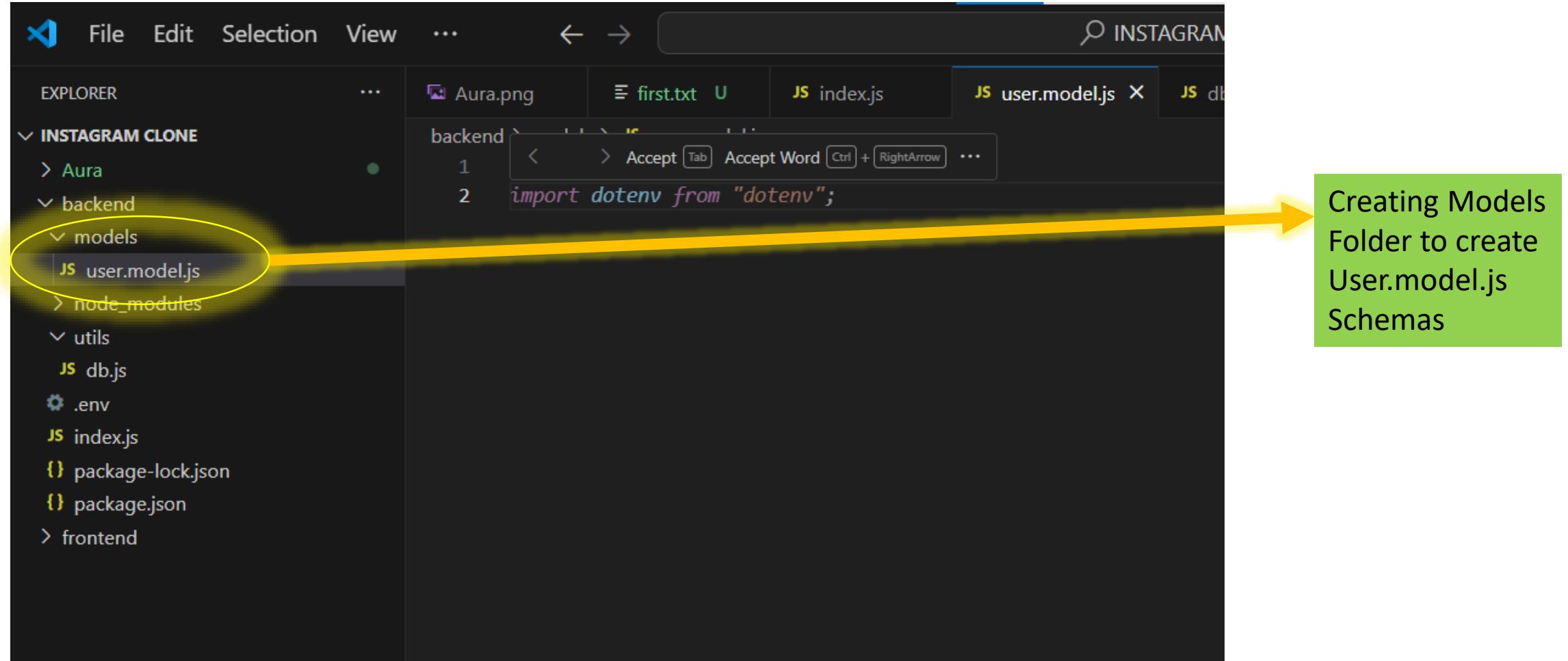
This function will connect
To the mongodb using
url

```
63
64
65
66 app.listen(PORT, ()=>{
67   connectDB();
68   console.log(`server is running on port ${PORT}`);
69 })
```

As the server is running..
So we will write connect db() there

```
Node.js v22.16.0
[nodemon] app crashed - waiting for file changes before starting...
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
[dotenv@17.2.2] injecting env (2) from .env -- tip: 🚀 auto-backup env with Radar: https://dotenvx.com/radar
server is running on port 5000
MongoDB connected
 
🔍 Ln 57, Col 1
```

Now we will create models and schema for db



Creating the userSchema

```
1 import mongoose from "mongoose";
2
3 const userSchema = new mongoose.Schema({
4   username:{type:String, required:true, unique:true},
5
6   email:{type:String, required:true, unique:true},
7
8   password:{type:String, required:true},
9
10  profilePicture:{type:String, default:""},
11
12  bio:{type:String, default:""},
13
14  gender:{type:String, enum:['male','female']},
15
16  following:[
17    {
18      type:Array, default:[]
19    }
20  ],
21  isAdmin:{type:Boolean, default:false}
22 },{timestamps:true
23 })
24
25
```

Creating the user Schema
With new mongoose.Schema

This is the username

Here we have used enum
Basically gender me 2 values
Rakhege..and user choose from it

```
11
12   bio:{type:String, default:""},  

13
14   gender:{type:String, enum:['male','female']},  

15
16   followers:{type:mongoose.Schema.Types.ObjectId, ref:'User'},  

17 //ye jo followers hai..wo user model ko hi refer kr rhe hai  

18
19 //this mongoose.Schema.Types.ObjectId is used to create a reference  

20 // another document in a different collection  

21
22 //this User is the name of the model we are referencing  

23
24   following:{type:mongoose.Schema.Types.ObjectId, ref:'User'},  

25
26 posts:[{type:mongoose.Schema.Types.ObjectId, ref:'Post'}],  

27 //we will create a schema of post..so the relation is with post  

28
29 bookmarks:[{type:mongoose.Schema.Types.ObjectId, ref:'Post'}],  

30
31
32 },{timestamps:true});  

33 //timestamps:true -- this will automatically create the createdAt  

34
35 export const User = mongoose.model("User", userSchema);
```

This mongoose.Schema.Types.ObjectId
This is used to create the relation with
Another document...
Basically uski id k through create karega

This timestamps:true ye basically mongodb me
createdAt and updatedAt filed create karega
Jissme hame pata lag jayega kab user aaya tha

This will create user model and export it and
We can perform crud operations on that user
model

Creating post.model.js Schema

The screenshot shows a dark-themed code editor interface. On the left is the Explorer sidebar with a tree view of the project structure:

- INSTAGRAM CLONE
 - Aura
 - backend
 - models
 - post.model.js (selected)
 - user.model.js
 - node_modules
 - utils
 - db.js
 - .env
 - index.js
 - package-lock.json
 - package.json
- frontend

The main editor area displays the content of the selected file, post.model.js:

```
backend > models > post.model.js > ...
1 import mongoose from "mongoose";
2 const postSchema = new mongoose.Schema({
3   caption:{type:String, default:""},
4   //instagram pr post upload karte waqt caption optional hota hai
5
6   image:{type:String, required:true},
7   //image is required..agr image nhi di to post nhi hoga
8
9   author:{type:mongoose.Schema.Types.ObjectId, ref:'User', required:true},
10  //ye author field user model ko refer kr rha hai
11  //jab post create karega..to batayege konse user ne create kara hai use
12
13  likes:[{type:mongoose.Schema.Types.ObjectId, ref:'User'}],
14  //like krne wala user hi hoga
15
16  comments:[{type:mongoose.Schema.Types.ObjectId, ref:'Comment'}]
17  //we will create another model of comment..so the relation is with comment model
18  //basically comment me bhi proper ye hota hai kisne comment kiya...
19
20
21 })
22 export const Post = mongoose.model("Post", postSchema);
```

Creating comment Schema

The screenshot shows a code editor interface with the title bar "INSTAGRAM CLONE". The left sidebar displays a file tree for an "INSTAGRAM CLONE" project, including files like "Aura.png", "first.txt", "index.js", "user.model.js", "post.model.js", "comment.model.js", "db.js", ".env", and "frontend". The main editor area shows the content of "comment.model.js". The code defines a Mongoose schema for a "comment" model:

```
import mongoose from "mongoose";
const commentSchema = new mongoose.Schema({
  text:{type:String, required:true},
  //comment ka text hona chahiye..ye required hai
  author :{type:mongoose.Schema.Types.ObjectId, ref:'User', required:true},
  //ye author field user model ko refer kr rha hai
  //this has advance relations like one to one , many type
  post :{type:mongoose.Schema.Types.ObjectId, ref:'Post', required:true},
})
export default commentSchema = mongoose.model('Comment', commentSchema);
```

- Now till now we have developed basic social media model
- Now we are developing chat model
- Which has two schemas
- Kisne message kiya aur kya kya baate chal rahi hai



File Edit Selection View ...

← →

INSTAGRAM CLONE

EXPLORER

INSTAGRAM CLONE

> Aura

backend

models

JS comment.model.js

JS conversation.model.js

JS message.model.js

JS post.model.js

JS user.model.js

> node_modules

utils

JS db.js

.env

JS index.js

index.js

JS user.model.js

JS post.model.js

JS comment.model.js

JS message.model.js X

backend > models > JS message.model.js > [?] default

```
1 import { mongo } from "mongoose";
2 const messageSchema = new mongoose.Schema({
3     senderId:{type:mongoose.Schema.Types.ObjectId, ref:'User'},
4     //we will keep the sender id...
5     //and this is obvious..jo message send karne wala hoga..vo ek user hi hoga..
6
7     receiverId:{type:mongoose.Schema.Types.ObjectId, ref:'User'},
8     message:{type:String, required: true}
9 })
10 export default Message = mongoose.model('Message',messageSchema);
```

Now we have created Message model which has sender id , receiver id and message

INSTAGRAM CLONE

EXPLORER

INSTAGRAM CLONE

- > Aura
- backend
- models
 - JS comment.model.js
 - JS conversation.model.js**
 - JS message.model.js
 - JS post.model.js
 - JS user.model.js
- > node_modules
- utils
 - JS db.js
 - .env
- JS index.js
- { package-lock.json
- { package.json
- > frontend

index.js user.model.js post.model.js comment.model.js message.model.js conversation.model.js

backend > models > JS conversation.model.js > [?] default

```
1 import mongoose from "mongoose";
2 const conversationsSchema = new mongoose.Schema({
3     //conversation me participants involve hote hai
4     participants:[
5         type:mongoose.Schema.Types.ObjectId,
6         ref : 'User'
7             //now it can have one or multiple users..so we just create array
8     ],
9     //participants se ye hoga ki example k liye 2 user hai..to unke bich k saare message ki
10    //list chahiye mujhe
11
12    message : [
13        type: mongoose.Schema.Types.ObjectId,
14        ref : 'Message'
15        //
16    ]
17
18 })
19 export default conversationsSchema = mongoose.model['Conversation' , conversationsSchema];
```

Now this we have created conversation model

Now till now the database work is done

- //now we are creating controllers folder
- First we are creating the register function in which the user register

```
1 import { User, User } from "../models/user.model";
2
3 export const register = async (req, res) => {
4   try {
5     //register karne ke liye username , email and password
6     const {username, email, password} = req.body;
7     //req.body.. jb bhi ham form submit krte hai..
8     //to us data ko ham req.body se retrieve kar sakte hai
9
10
11    //this below if statement to check that the user has properly entered details or not
12    if(!username || !email || !password){
13      //this means error
14      return res.status(401).json({
15        message: "Something is missing",
16        success: false,
17      })
18    }
19
20  }
```

Req.body – when we do form submission
Then the data is in req ...
We can also destructure it with
Req.body.username

This is the default thing..
basically
We are checking whether we have
Received the username , password
And email properly or not

```
//now we will check if the user has already registered or not
const user = await User.findOne({email}),
//this findOne method will check that the user exist in database or not

if(user){
  return res.status(401).json({
    message: "The email already exists... Try with Different Email",
    success: false, //kaam abhi bhi pura nahi hua
  })
}
```

findOne
This will check if email exist in the
Mongodb or not

Bcrypt --- Hashing Password

```
import bcrypt from "bcryptjs";
```

[**Npm I bcrypt**](#)

```
//  
const hashedPassword = await bcrypt.hash(password,10);  
const User = await User.create({  
    username,  
    password : hashedPassword, //now we need to hash password using bcrypt  
  
    email,  
    //here we need to enter those filed which are necessary (required : true)  
});
```

bcrypt.hash(password, 10)
takes the plain password
and a "salt rounds" value
(10 here).

"Salt rounds" means how many times bcrypt will process the password
to make the hash stronger. More rounds = more secure, but slower.

Bcrypt – compare method

```
const isPasswordMatch = await bcrypt.compare(password,user.password);  
//this method will compare the password enter by the user in the database
```



This method will simply check the password which the user has entered
Equals to the available password in the database or not

Token

- After checking that the user is authenticated or not
- // now we generate tokens
- Token is stored in cookie...until the user is logged in the site
- The token is stored in cookie...
- Basically cookie me token agar store hai..to user logged in hai

- Token has a time limit – after that ..we will again log in the user

Token

This will create a signed token

```
const token = await jwt.sign({userId: user._id}, process.env.SECRET_KEY, {expiresIn: '1d'});
```

Payload : it stores the user Unique id in token

process.env.SECRET_KEY is the secret key used to sign (encrypt) the token.
Only your server should know this key.

process.env.SECRET_KEY is the secret key used to sign (encrypt) the token.
Only your server should know this key.

```
return res.cookie("token", token, {  
    httpOnly: true,  
    sameSite: 'strict',  
    maxAge: 24 * 60 * 60 * 1000, //  
    //this maxage is in milliseconds  
    //24 hours * 60 minutes * 60 seconds  
    //this will not allow the client  
    //this is for security purpose
```

res.cookie("token", token, {...}) sets a cookie named "token" with the JWT value.

httpOnly: true means JavaScript running in the browser cannot access this cookie (for security).

sameSite: 'strict' helps prevent CSRF attacks by only sending the cookie for same-site requests.

```
}).json({  
    message: `Login Successful ${user.username}`,  
    success: true,  
    user  
});
```

```
user = {  
    _id: user._id,  
    username: user.username,  
    email: user.email,  
    profilePicture: user.profilePicture,  
    bio: user.bio,  
    followers: user.followers,  
    following: user.following,  
    posts: user.posts,  
}
```

Logout

simply set empty token ,
maxage = 0

```
export const logout = (req,res)=>{
  try{

    //logout is simply..we just simply delete the token from the cookie..
    //just replace it with empty string
    return res.cookie("token","", {maxAge:0}).json({
      message:"Logged out successfully",
      success:true,
    })
  }catch(error){
    console.log(error);
  }
}
```

getProfile function

```
//instagram pr get profile wala bhi page hota hai
export const getProfile = async(req,res)=>{
    try{
        const userId = req.params.userId;
        const user = await User.findById(userId);
        if(!user){
            return res.status(404).json({
                message:"User not found",
                success:false,
            })
        }
        return res.status(200).json({
            message:"User profile fetched successfully",
            success:true,
            user
        })
    }
    catch(error){
        console.log(error);
    }
}
```

Middleware to check authentication

```
const isAuthenticated = async (req , res , next) =>{
  try{
    const token = req.cookie.token; //req.cookie ke andar token hota hai

    //first we check if the user is authenticated or not...
    if(!token) return res.status(401).json({
      message:"User not authenticated",
      success: false,
    });

    //now we have found the user
    //we will verify again with the secret key in our environment variables

    const decode = await jwt.verify(token,process.env.SECRET_KEY);
    //now from here it will verify

    if(!decode){
      return res.status(401).json({
        message:"User not authenticated",
        success: false,
      })
    }

    //now till now the token is decoded..verified
    req.id = decode.userId;

    next();
  }
  catch(error){
    console.log(error);
  }
}
```

it tries to get token from the cookie send by the client
Cookie k andar token hoga

It verifies the token using your secret key. If the token is valid, it decodes the payload (which contains user info).

If the token is valid, it adds the user's ID from the token to the request object, so you can use it in the next middleware or route handler.

It will push to next middleware

Cloudinary

backend > utils > **JS** clouddinary.js > ...

```
1 import {v1 as cloudinary} from 'cloudinary';
2 import dotenv from 'dotenv';
3 dotenv.config();

4 // Configure Cloudinary with your credentials
5 cloudinary.config({
6   cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
7   api_key: process.env.CLOUDINARY_API_KEY,
8   api_secret: process.env.CLOUDINARY_API_SECRET
9 });
10 export default cloudinary;
11
```

import {v1 as cloudinary} from 'cloudinary';
Imports the v1 API from the
cloudinary package and
renames it to cloudinary for use in your
code.

this will config the
cloudinary with the
credentials from the
.env file

datauri

```
import DataUriParser from "datauri/parser.js";
```

npm i datauri parser

A Data URI is a way to **embed file data** (like images or documents) **directly in a string**, usually for use in HTML, CSS, or when uploading files to cloud services.

For example, it can turn an image file into a string like:
data:image/png;base64,iVBORw0KGgoAAAANSUhEUg...

backend > utils > JS datauri.js > [o] default

```
1 import DataUriParser from "datauri/parser.js";
2 import path from "path";
3
4
5 const parser = new DataUriParser(); -----> Create instance
6
7 //it converts file or buffers to string
8
9 const getDataUri = (file) => {
10   const extName = path.extname(file.originalname).toString();
11   return parser.format(extName, file.buffer);
12 }
13 export default getDataUri;
```

The **datauri parser** (often used via the `datauri` or `datauri/parser` npm package) is a utility that converts files or buffers into Data URI format. A Data URI is a way to embed file data (like images or documents) directly in a string, usually for use in HTML, CSS, or when uploading files to cloud services.

For example, it can turn an image file into a string like:
`data:image/png;base64,iVBORw0KGgoAAAANSUhEUg...`

EditProfile – Cloudinary and dataUri

```
export const editProfile = async(req,res)=>{
  try{
    const userId = req.id;
    //now here we will set up the cloudinary
    const {bio,gender}= req.body;
    const profilePicture = req.file;
    let cloudResponse;

    if(profilePicture){
      const dataUri = getDataUri(profilePicture);
      cloudResponse = await cloudinary.uploader.upload(dataUri);
    }

    const user = await User.findById(userId);
    if(!user){
      return res.status(404).json({
        message:"User not found",
        success:false,
      })
    }
    if(bio)user.bio = bio;
    if(gender)user.gender = gender;
    if(cloudResponse){
      user.profilePicture = cloudResponse.secure_url;
    }
    await user.save();
    return res.status(200).json({
      message:"Profile updated successfully",
      success:true,
      user
    })
  }
}
```

DataUri se profile picture ko string me Convert kiya

This is the way of uploading in cloudinary

cloudResponse.secure_url contains the direct, secure (https) link to the image you just uploaded to Cloudinary.
user.profilePicture = cloudResponse.secure_url saves this URL in the user's profile in your database

If the User update any field..
So It will update that in database

getSuggestedUser

```
//now instagram also has the feature of suggesting people..so let us build that
export const getSuggestedUsers = async(req,res)=>{
  try{
    const SuggestedUsers = await User.find({_id:{$ne:req.id}}).select("-password").limit(10);

    if(!SuggestedUsers){
      return res.status(404).json({
        message:"No user found",
        success:false,
      })
    }
    return res.status(200).json({
      message:"Suggested users fetched successfully",
      success:true,
      user: SuggestedUsers,
    })
  }
  catch(error){
    console.log(error);
  }
}
```

This will make sure that only 10 suggested user is shown

This will simply remove the password field (for safety)

It will find the id of the user from database which is not equals to admin id

Follow or unfollow function

```
export const followOrUnfollowUser = async(req,res)=>{
  try{
    const followkarnewala = req.id; //logged in user //ye authenticated user
    const jiskofollowkruga = req.params.id; //the user to be followed or unfollowed

    if(followkarnewala === jiskofollowkruga){
      return res.status(400).json({
        message:"You cannot follow/unfollow yourself(Self obsessed)",
        success:false,
      })
    }

    const user = await User.findById(followkarnewala); //this is me .//the user
    const targetUser = await User.findById(jiskofollowkruga); //this is the user whom i want to follow

    //now we will need to check for error
    if(!user){
      return res.status(404).json({
        message:"User not found",
        success:false,
      })
    }

    if(!user || !targetUser){
      return res.status(404).json({
        message:"User not found",
        success:false,
      })
    }
  }
}
```

This is authenticated user id ..
For example my(admin) id

This is the id of the person whom I Will follow , and will be received in params

Now I will find the id in Mongodb database

Again doing the same

```
// now we check follow krdha hai ya nahi
const isFollowing = user.following.includes(jiskofollowkruga); //this is array
//like if i follow someone..then it will add in my following array
//agar follow kar rakha hai..to true return kardega
//if follow kar rakha hai..to unfollow kardega

if(isFollowing){
    //unfollow
    await Promise.all([
        User.updateOne({_id: followkarnewala}, {$pull: {following: jiskofollowkruga}}),
        //user k andar jo uska following array hai..usme jisko follow karna hai..uska id push kardega

        User.updateOne({_id: jiskofollowkruga}, {$pull: {followers: followkarnewala}})
        //now jisko mene follow kara hai..uske followers array ko bhi update karna hai
    ]);

    return res.status(200).json({
        message: "Unfollowed successfully",
        success: true,
    })
}
else{
    await Promise.all([
        User.updateOne({_id: followkarnewala}, {$push: {following: jiskofollowkruga}}),
        //user k andar jo uska following array hai..usme jisko follow karna hai..uska id push kardega

        User.updateOne({_id: jiskofollowkruga}, {$push: {followers: followkarnewala}})
        //now jisko mene follow kara hai..uske followers array ko bhi update karna hai
    ]);

    return res.status(200).json({
        message: "Followed successfully",
        success: true,
    })
}

}
```

it runs multiple database updates in parallel using `Promise.all`:

Add

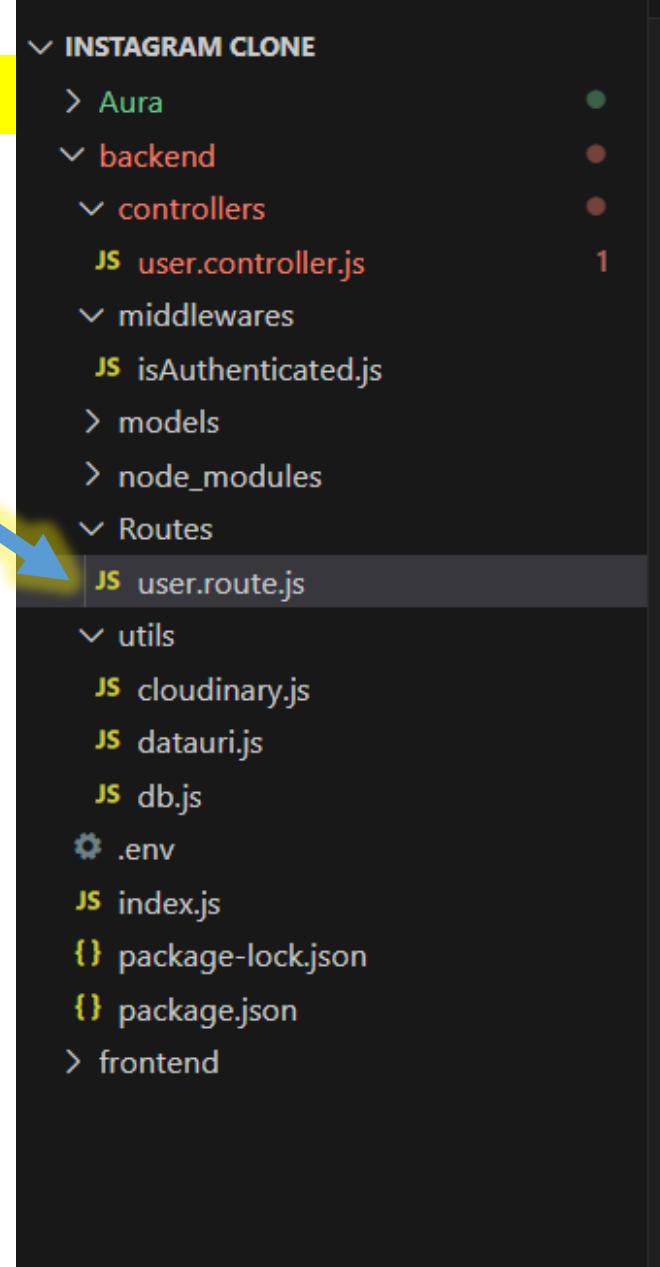
Removes the logged-in user's ID from the target user's followers array.

Routes

Now after making the folder structure..we are making a new folder for defining the routes

```
1 import express from "express";
2 import {register, login, getUser, updateUser, followAndUnfollow, getProfile, editProfile, getSuggestedUsers} from "../controllers/user.controller";
3 import isAuthenticated from "../middlewares/isAuthenticated.js";
4 const upload = require("../middlewares/multer.js");
5
6 const router = express.Router();
7
8 router.route("/register").post(register);
9
10 router.route("/login").post(login);
11
12 router.route("/logout").get(logout);
13
14 router.route("/:id/profile").get(isAuthenticated, getProfile);
15
16 router.route("/profile/edit").post(isAuthenticated, upload.single("profilePicture"), editProfile);
17
18 router.route("/suggested").get(isAuthenticated, getSuggestedUsers);
19
20 router.route("/followorunfollow/:id").post(isAuthenticated, followAndUnfollow);
21
22 export default router;
```

multer



Multer

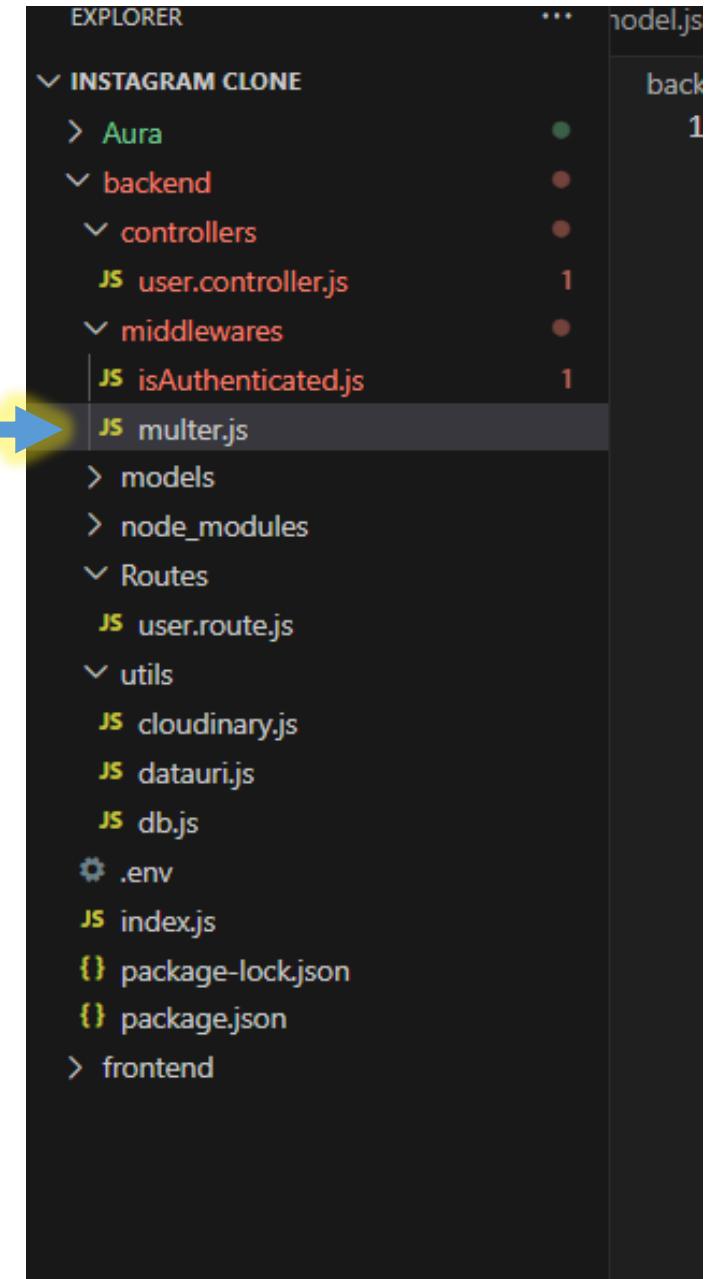
npm i multer

```
● PS C:\Users\DELL\Desktop\INSTAGRAM CLONE> cd backend
● PS C:\Users\DELL\Desktop\INSTAGRAM CLONE\backend> npm i multer

added 17 packages, and audited 155 packages in 3s

23 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\DELL\Desktop\INSTAGRAM CLONE\backend>
```



Multer

```
backend > middlewares > js multer.js > [e] default  
1 import multer from "multer";  
2 const upload = multer({  
3   storage: multer.memoryStorage(),  
4 })  
5 export default upload;
```

storage: `multer.memoryStorage()` tells multer to store uploaded files in memory as Buffer objects (not on disk).

Exports the configured upload middleware so you can use it in your routes to handle file uploads.

Here we have created the routers

```
EXPLORER          ... model.js   JS comment.model.js   JS message.model.js   JS conversation.model.js   JS user.controller.js 1   JS user.route.js X   JS multer.js   JS cloudinary.js   JS datauri.js   JS isAuthenticated.js  
INSTAGRAM CLONE > Aura  
backend > Routes > JS user.route.js > ...  
controllers > user.controller.js  
middlewares > isAuthenticated.js  
multer.js  
models  
node_modules  
Routes > JS user.route.js  
utils > cloudinary.js  
datauri.js  
db.js  
.env  
index.js  
package-lock.json  
package.json  
frontend
```

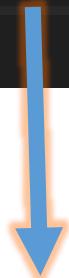
```
1 import express from "express";
2 import {register, login, getUser, updateUser, followAndUnfollow, getProfile, editProfile, getSuggestedUsers} from "../controllers/user.controller.js";
3 import isAuthenticated from "../middlewares/isAuthenticated.js";
4 const upload = require("../middlewares/multer.js");
5
6 const router = express.Router();
7
8 router.route("/register").post(register);
9
10 router.route("/login").post(login);
11
12 router.route("/logout").get(logout);
13
14 router.route("/:id/profile").get(isAuthenticated, getProfile);
15
16 router.route("/profile/edit").post(isAuthenticated, upload.single("profilePicture"), editProfile);
17
18 router.route("/suggested").get(isAuthenticated, getSuggestedUsers);
19
20 router.route("/followorunfollow/:id").post(isAuthenticated, followAndUnfollow);
21
22 export default router;
23
```

Multer used here

Creating First API

```
import userRoute from "./Routes/user.route.js";
```

```
app.use("/api/v1/user",userRoute);
```



means that all routes defined in userRoute will be accessible under the path /api/v1/user

if userRoute has a route for /login, it will be available at /api/v1/user/login.

<http://localhost:8000/api/v1/user/register>

ThunderClient– Testing the api

The screenshot shows the ThunderClient interface for testing APIs. On the left, the request details are set: method **POST**, URL **http://localhost:5000/api/v1/user/register**, and body type **Body 1**. The body content is JSON:

```
1 {
2   "username": "sparshsharma",
3   "email" : "sparsh@gmail.com",
4   "password": "12345"
5 }
```

A blue arrow points from the **Send** button at the top right towards the JSON content area. Another blue arrow points from the **Response** tab on the right towards the response content. A yellow box at the bottom center contains the text **Register api working**.

On the right, the response details are displayed: **Status: 201 Created**, **Size: 57 Bytes**, and **Time: 429 ms**. The **Response** tab is selected, showing the JSON response:

```
1 {
2   "message": "Account created Successfully",
3   "success": true
4 }
```

Register api working

Overview

DATABASES: 1 COLLECTIONS: 1

PREVIEW

New Data Explorer

VISUALIZE YOUR DATA

REFRESH

DATABASE

+ Create Database

Clusters

SERVICES

Atlas Search

Stream Processing

Triggers

Migration

Data Federation

SECURITY

Quickstart

Backup

Database Access

Network Access

Advanced

Goto

Search Namespaces

test

users

test.users

STORAGE SIZE: 20KB LOGICAL DATA SIZE: 268B TOTAL DOCUMENTS: 1 INDEXES TOTAL SIZE: 60KB

Find

Indexes

Schema Anti-Patterns

Aggregation

Search Indexes

Generate queries from natural language in Compass

INSERT DOCUMENT

Filter

Type a query: { field: 'value' }

Reset

Apply

Options

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('68d37e416d154171a2ebc9e7')
username : "sparshsharma"
email : "sparsh@gmail.com"
password : "$2b$10$BeJT3s2fR4Sz38JwEW6GeA/SE7nZ2fAyPbVJFgNoGSnMvUEbAfxK"
profilePicture : ""
bio : ""
posts : Array (empty)
bookmarks : Array (empty)
createdAt : 2025-09-24T05:14:41.812+00:00
updatedAt : 2025-09-24T05:14:41.812+00:00
__v : 0
```

Hi, how can I help you get started?
MBot • 17h ago

Need help creating my account

Is Atlas right for me? Something else

Data updating



Custom Status: All Good



The screenshot shows the Thunder Client extension integrated into the Visual Studio Code interface. The top navigation bar includes tabs for 'isAuthenticated.js', 'db.js', '.env', and several 'New Request' and 'localhost:3000/' entries. On the left, there's a sidebar with icons for file operations, search, and activity, along with a list of recent requests.

A context menu is open over a 'POST' request to 'localhost:5000/api/v1/user/register' made 5 minutes ago. The menu options include 'Run Request', 'Save to Collection', 'Open in New Tab', 'Rename', 'Duplicate', and 'Delete'. The 'Save to Collection' option is highlighted.

The main area displays the 'Add To Collection' dialog. It contains fields for 'Request Name' (set to 'Register'), 'Url' (set to 'http://localhost:5000/api/v1/user/register'), 'Collection' (set to 'Create New'), and 'New Name' (set to 'Aura'). A success message at the bottom states 'Saved successfully, You can close window now'.

At the bottom of the screen, the VS Code status bar shows tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is active), and 'PORTS'. There are also standard VS Code window control buttons on the right.

Login api testing

The screenshot shows the Postman application interface for testing a login API.

Request Section:

- Method: POST
- URL: `http://localhost:5000/api/v1/user/login`
- Headers: 2
- Auth: None
- Body**: 1 (highlighted)
- Tests: None
- Pre Run: None

Body Content (JSON):

```
1 {  
2   "email" : "sparsh@gmail.com",  
3   "password": "12345"  
4 }  
5 
```

Response Section:

Status: 200 OK | Size: 0 Bytes | Time: 137 ms

Response Headers (10):

- Content-Type: application/json
- Content-Length: 0
- Connection: keep-alive
- Date: Mon, 10 Jul 2023 10:30:00 GMT
- Server: Werkzeug/2.1.2 Python/3.11.3
- X-Powered-By: Gunicorn/20.1.1
- Access-Control-Allow-Origin: *
- Access-Control-Allow-Methods: POST, GET, PUT, DELETE, PATCH, OPTIONS
- Access-Control-Allow-Headers: Content-Type, Authorization, X-CSRFToken

Cookies (1):

- session_id: 68d37e416d154171a2ebc9e7

Results:

```
1 {  
2   "message": "Login Successful sparshsharma",  
3   "success": true,  
4   "user": {  
5     "_id": "68d37e416d154171a2ebc9e7",  
6     "username": "sparshsharma",  
7     "email": "sparsh@gmail.com",  
8     "profilePicture": "",  
9     "bio": "",  
10    "posts": []  
11  }  
12 }  
13 
```

token

The screenshot shows the Postman interface with a yellow callout pointing to the word "token".

Request (Left Panel):

- Method: POST
- URL: `http://localhost:5000/api/v1/user/login`
- Body tab is selected.
- JSON Content:

```
1 {
2
3     "email" : "sparsh@gmail.com",
4     "password": "12345"
5 }
```

Response (Right Panel):

- Status: 200 OK
- Size: 0 Bytes
- Time: 137 ms
- Cookies tab is selected.
- Request Url: `http://localhost:5000/api/v1/user/login`
- Response Cookies table:

Name	Value
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2OGQzM2U0MTZkMTU0MTcxYTJlYmM5ZTciLCJpYXQiOjE3NTg2OTExOTIwMzV4cCI6MTc1ODc3NzU5Mn0.Wj7ey92G0r-_KqTM40lrkfINn9jwM0zSYG4WAg0QI38

Logout api testing

The screenshot shows the Postman application interface for testing an API endpoint. The left panel displays the request configuration, and the right panel shows the response details.

Request Configuration (Left Panel):

- Method: GET
- URL: `http://localhost:5000/api/v1/user/logout`
- Body tab is selected.
- Format: JSON
- JSON Content:

```
1 {  
2  
3   "email" : "sparsh@gmail.com",  
4   "password": "12345"  
5 }
```

Response Details (Right Panel):

- Status: 200 OK
- Size: 0 Bytes
- Time: 5 ms
- Response tab is selected.
- Response Body:

```
1 {  
2   "message": "Logged out successfully",  
3   "success": true  
4 }
```

Getting the user details using id

The screenshot shows a POSTMAN API client interface. The top bar displays the method as 'GET' and the URL as 'http://localhost:5000/api/v1/user/68d37e416d154171a2ebc9e7/profile'. A 'Send' button is visible. Below the URL, there are tabs for 'Query', 'Headers 2', 'Auth', 'Body 1', 'Tests', and 'Pre Run'. The 'Body 1' tab is selected, showing the JSON content sent in the request body:

```
1 {  
2     "email" : "sparsh@gmail.com",  
3     "password": "12345"  
4 }  
5 
```

The 'Format' dropdown next to the JSON content is set to 'JSON'. To the right, the response section shows the status as '200 OK', size as '298 Bytes', and time as '1.26 s'. The 'Response' tab is selected, displaying the following JSON output:

```
1 {  
2     "message": "User profile fetched successfully",  
3     "success": true,  
4     "user": {  
5         "_id": "68d37e416d154171a2ebc9e7",  
6         "username": "sparshsharma",  
7         "email": "sparsh@gmail.com",  
8         "profilePicture": "",  
9         "bio": "",  
10        "posts": [],  
11        "bookmarks": [],  
12        "createdAt": "2025-09-24T05:14:41.812Z",  
13        "updatedAt": "2025-09-24T05:14:41.812Z",  
14        "__v": 0  
15    }  
16 }  
17 
```

At the bottom right, there are buttons for 'Response' (which is highlighted in blue), 'Chart', and a magnifying glass icon.

Edit profile api testing with sending data

GET Send

Query Headers ² Auth **Body** ¹ Tests Pre Run

JSON XML Text **Form** Form-encode GraphQL Binary

Form Fields Files Import

<input checked="" type="checkbox"/> bio	this is bio of person
<input checked="" type="checkbox"/> gender	male
<input type="checkbox"/> field name	value

Files profilePicture Choose File Aura.png

<input type="checkbox"/> field name	Choose File Select file
-------------------------------------	-------------------------

Learn more about how to set a custom [content-type](#) for a field.

Status: 200 OK Size: 298 Bytes Time: 1.26 s

Response Headers ⁹ Cookies Results Docs { }

```
1  {
2    "message": "User profile fetched successfully",
3    "success": true,
4    "user": {
5      "_id": "68d37e416d154171a2ebc9e7",
6      "username": "sparshsharma",
7      "email": "sparsh@gmail.com",
8      "profilePicture": "",
9      "bio": "",
10     "posts": [],
11     "bookmarks": [],
12     "createdAt": "2025-09-24T05:14:41.812Z",
13     "updatedAt": "2025-09-24T05:14:41.812Z",
14     "__v": 0
15   }
16 }
```

Response Chart

The screenshot shows the Thunder Client interface with the following details:

- Header Bar:** Shows tabs for various files like `user.route.js`, `multer.js`, `cloudinary.js`, `datauri.js`, and `isAuthenticated.js`. The status bar indicates "Extension: Thunder Client" and a GET request to "localhost:5000:api/v1/use..".
- Request Section:** A POST request is being made to `http://localhost:5000/api/v1/user/profile/edit`. The "Body" tab is selected, showing the following data:
 - Form:** Contains fields: "bio" (checked, value: "this is bio of person"), "gender" (checked, value: "male"), and an unchecked field "field name" (value: "value").
 - Files:** Contains a checked file "profilePicture" (value: "Aura.png") and an unchecked field "field name" with a "Choose File" button (value: "Select file").
- Response Section:** The response is a 200 OK status with 404 Bytes and 111 ms time. The response body is a JSON object:

```
1  {
2      "message": "Profile updated successfully",
3      "success": true,
4      "user": {
5          "_id": "68d37e416d154171a2ebc9e7",
6          "username": "sparshsharma",
7          "email": "sparsh@gmail.com",
8          "password": "$2b$10$BeJT3s2fR4Szz38JwEW6GeA
9              /SE7nZ2fAyPbVJFgNoGSnMvUEbAfxK",
10         "profilePicture": "",
11         "bio": "this is bio of person",
12         "posts": [],
13         "bookmarks": [],
14         "createdAt": "2025-09-24T05:14:41.812Z",
15         "updatedAt": "2025-09-24T06:04:20.695Z",
16         "__v": 0,
17         "gender": "male"
18     }
19 }
```
- Bottom Navigation:** Includes links for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (selected), PORTS, and a node - backend terminal.
- Terminal:** Shows the command `PS C:\Users\DELL\Desktop\INSTAGRAM CLONE\backend> npm run dev`.

Suggested api testing

The screenshot shows a REST API testing interface with the following details:

Request: GET http://localhost:5000/api/v1/user/suggested

Body: Form (selected)

Form Fields:

- bio: this is bio of person
- gender: male
- field name: value

Files:

- profilePicture: Choose File (selected), Aura.png
- field name: Choose File, Select file

Response: Status: 200 OK, Size: 75 Bytes, Time: 40 ms

```
1 {
2   "message": "Suggested users fetched successfully",
3   "success": true,
4   "user": []
5 }
```

Copy button is available for the response JSON.

Learn more about how to set a custom [content-type](#) for a field.

Buttons at the bottom: Response, Chart, and a copy icon.

GET Send

Query Headers ² Auth **Body ¹** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 []
2
3   "username" : "sparsh",
4   "email" : "sparshsharma@gmail.com",
5   "password": "12345"
6 []|
```

Status: 200 OK Size: 303 Bytes Time: 42 ms

Response Headers ⁹ Cookies Results Docs { } =

```
1 {
2   "message": "Suggested users fetched successfully",
3   "success": true,
4   "user": [
5     {
6       "_id": "68d38d86972ab93f9c00f878",
7       "username": "sparsh",
8       "email": "sparshsharma@gmail.com",
9       "profilePicture": "",
10      "bio": "",
11      "posts": [],
12      "bookmarks": [],
13      "createdAt": "2025-09-24T06:19:50.922Z",
14      "updatedAt": "2025-09-24T06:19:50.922Z",
15      "__v": 0
16    }
17  ]
18 }
```

Follow and unfollow api testing

The screenshot shows a POST request to `http://localhost:5000/api/v1/user/followorunfollow/68d37e416d154171a2ebc`. The response status is 200 OK, size is 50 Bytes, and time is 456 ms. The response body is:

```
1 {  
2   "message": "Followed successfully",  
3   "success": true  
4 }
```

The screenshot shows a POST request to `http://localhost:5000/api/v1/user/followorunfollow/68d37e416d154171a2ebc`. The response status is 200 OK, size is 52 Bytes, and time is 134 ms. The response body is:

```
1 {  
2   "message": "Unfollowed successfully",  
3   "success": true  
4 }
```

Now till now we have build and tested the api

now we are building the image uploader

Sharp

```
import sharp from 'sharp';
```

Sometimes the images which we upload..its quality is too high
So in order to **optimize** it ..we will have to use **sharp --- a library**

npm i sharp

```
PS C:\Users\DELL\Desktop\INSTAGRAM CLONE> cd backend
PS C:\Users\DELL\Desktop\INSTAGRAM CLONE\backend> npm i sharp

added 4 packages, and audited 159 packages in 16s

25 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\DELL\Desktop\INSTAGRAM CLONE\backend>
```

Now we are building post model using sharp, and more

```
import sharp from 'sharp';
import Post from '../models/post.model.js';
import { v2 as cloudinary } from 'cloudinary';
import { User } from '../models/user.model.js';

export const addNewPost = async (req, res) => {
  try{
    const {caption} = req.body;
    const image = req.file;

    const authorId = req.id; //jo log in user hoga uski id
    if(!image){
      return res.status(400).json({
        message:"image is required",
        success:false,
      })
    }
    //now image upload
    //we have to use multer

    //sometimes image quality and size is very large --- so we use npm sharp
    //inside sharp..we have to pass the buffer value of the image
    const optimizedImage = await sharp(image.buffer)
      .resize({width:800, height:800,fit:'inside'}).jpeg({quality:80}).toBuffer();

    //now we have to convert this buffer to datauri
    const fileUri = `data:image/jpeg;base64,${optimizedImage.toString('base64')}`;

    const cloudResponse = await cloudinary.uploader.upload(fileUri);
    //ye cloudResponse ke andar hume url milega

    const post = await Post.create({
      caption,
      image:cloudResponse.secure_url,
      author : authorId,
    })

    //now jaise hi koi naya user koi nayi post create karga..so we need to push that also
    const user = await User.findById(authorId);
    if(user){
      user.posts.push(post._id);
      await user.save();
    }

    //now jaise hi koi naya user koi nayi post create karga..so we need to push that also
    const user = await User.findById(authorId);
    if(user){
      user.posts.push(post._id);
      await user.save();
    }

    //sometimes image quality and size is very large --- so we use npm sharp
    //inside sharp..we have to pass the buffer value of the image
    const optimizedImage = await sharp(image.buffer)
      .resize({width:800, height:800,fit:'inside'}).jpeg({quality:80}).toBuffer();

    //now we have to convert this buffer to datauri
    const fileUri = `data:image/jpeg;base64,${optimizedImage.toString('base64')}`;

    const cloudResponse = await cloudinary.uploader.upload(fileUri);
    //ye cloudResponse ke andar hume url milega

    const post = await Post.create({
      caption,
      image:cloudResponse.secure_url,
      author : authorId,
    })

    //now jaise hi koi naya user koi nayi post create karga..so we need to push that also
    const user = await User.findById(authorId);
    if(user){
      user.posts.push(post._id);
      await user.save();
    }

    await post.populate({path: 'author', select:'-password'});

    //now we need to do populate..
    //populate is a method in mongodb with which using the id we can get the complete details of the user

    return res.status(201).json([
      {
        message:"Post created successfully",
        success:true,
        post,
      }
    ])
  } catch(error){
    console.log(error);
    return res.status(500).json({
      message: "Internal server error",
      success: false,
    })
  }
}
```

Adding new post function

```
export const addNewPost = async (req, res) => {
  try{
    const {caption} = req.body;
    const image = req.file;

    const authorId = req.id;
    if(!image){
      return res.status(400).json({
        message:"image is required",
        success:false,
      })
    }

    const optimizedImage = await sharp(image.buffer)
      .resize({width:800, height:800, fit:'inside'}).jpeg({quality:80}).toBuffer();
  }
}
```

req → contains incoming request data (like body, files, headers, etc.)
res → used to send a response back to the client.

req.file → is set by **Multer**, which handles file uploads.

sharp(image.buffer)
it takes the image file's buffer(binary data)

.toBuffer()
returns the optimized image as a buffer again (for upload)

const fileUri = `data:image/jpeg;base64,\${optimizedImage.toString('base64')}`;

Converts the binary image buffer into a **Base64 string**.

```
const cloudResponse = await cloudinary.uploader.upload(fileUri);
```

Uploads the image to **Cloudinary** (an image hosting service).

```
const post = await Post.create({  
  caption,  
  image: cloudResponse.secure_url,  
  author: authorId,  
});
```

```
import mongoose from "mongoose";  
const postSchema = new mongoose.Schema({  
  caption: { type: String, default: "" },  
  image: { type: String, required: true },  
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },  
  likes: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],  
  comments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Comment' }]  
, { timestamps: true }  
export const Post = mongoose.model("Post", postSchema);
```

```
const { user } = useSelector(store=>store.auth);  
const { posts } = useSelector(store=>store.post);
```

```
const user = await User.findById(authorId);  
if(user){  
  user.posts.push(post._id);  
  await user.save();  
}
```

HTTPS link to the uploaded image on Cloudinary
storing that **image URL** in the database

Finds the user who created the post using their **ID (authorID)**
authorID comes from **req.id**

Saves the updated user document back to MongoDB

Adds the newly created post's ID to the user's posts array
In User model, there's a posts field that stores an array of post IDs
This will creates a **reference** between the user and their posts

```
await post.populate({path: 'author', select:'-password '});
```

This populate will allow to return all the fields

```
return res.status(201).json({  
  message:"Post created successfully",  
  success:true,  
  post,  
})
```

This means that the post is created

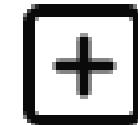
Create New Post



sparshsharma
Bio here...

Write a caption...

Select from computer



Create

```
const {user} = useSelector(store=>store.auth);  
  
const {posts} = useSelector(store=>store.post);
```

Each user has a posts array containing all their post IDs

When we highlight a post..then its username and other things are shown..so this is for that

```
export const getAllPosts = async (req, res) => {
  try{
    //this is nested populate

    const posts = await Post.find().sort({createdAt: -1}).populate({path: 'author', select: '-password'});//we want username and pro
.populate({path: 'author', select: 'username, profilePicture'})
.populate({
  path: 'comments',
  sort: {createdAt: -1},
  populate:{
    path: 'author',
    select: 'username profilePicture'
  }
});
//ye -1 isliye kr rhe hai..taaki latest post sabse upar aaye
//populate is used to get the complete details of the user using the id

    return res.status(200).json({
      message: "Posts fetched successfully",
      success: true,
      posts,
    })
  }
  catch(error){
    console.log(error);
    return res.status(500).json({
      message: "Internal server error",
      success: false,
    })
  }
}
```

Updated version of getAllPosts

```
export const getAllPosts = async (req, res) => {  
  try {
```

```
    const viewerId = req.id || null;
```

```
    const [publicUserIds, privateFollowedUserIds] = await Promise.all([  
      User.find({ isPrivate: false }).distinct('_id'),
```

Gives all the public user IDs

Finds all users who have isPrivate: false – means public account
.distinct('_id') – returns only the unique _id field values from the user

```
      viewerId ? User.find({ isPrivate: true, followers: viewerId }).distinct('_id') : Promise.resolve([]),  
    ]);
```

This ... is called spread operator

```
const allowedAuthorIds = [  
  ...publicUserIds,  
  ...(viewerId ? [viewerId] : []),  
  ...privateFollowedUserIds,  
];
```

Ids of all the public users

If user is logged in viewerId exist else it is null

Conditional Spread

Ids of all the private users who are in follow array

Gets the logged-in user's ID from the JWT token
If no user is logged in , viewerId is null

Runs two queries simultaneously

```
const posts = await Post.find({ author: { $in: allowedAuthorIds } })
```

Similar to
SELECT * FROM posts WHERE author

Finds posts where the **author** matches ANY ID in the **allowedAuthorIds**

```
.sort({ createdAt: -1 })
.populate({ path: 'author', select: 'username profilePicture isPrivate' })
.populate({
  path: 'comments',
  sort: { createdAt: -1 },
  populate: {
    path: 'author',
    select: 'username profilePicture'
  }
});
```

createdAt: -1 = **Descending order** (newest to oldest)

path: 'author' – means populate author field

select: 'username profilePicture isPrivate' → means **only return these fields** (and not things like password or email).

```
return res.status(200).json({
  message: "Posts fetched successfully",
  success: true,
  posts,
});
```

Nested populate

1. path: 'comments' – Populate the comments array
2. Sort : {createdAt: -1} – **sort comments newest first**
3. Populate : {path : 'author'} – for each comments also populate its author

JSON response back to the frontend with the fetched posts data.

getUserPost

Finds ALL posts where the author matches the specific user ID

```
//now this below one is for getting all the post of the user who is logged in  
export const getUserPost = async (req, res) => {  
  try {
```

```
    const authorId = req.id;
```

```
    const posts = await Post.find({ author: authorId }).sort({ createdAt: -1 }).populate({  
      path: 'author',  
      select: 'username, profilePicture'  
    }).populate({
```

```
      path: 'comments',  
      sort: { createdAt: -1 },  
      populate: {
```

```
        path: 'author',  
        select: 'username, profilePicture'  
      }
```

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

```
      posts,  
      success: true  
    })
```

```
  } catch (error) {  
    console.log(error);  
  }  
};
```

Sorts posts by creation date, newest first

// Before populate:
author: "user123"

// After populate:
author: {
 _id: "user123",
 username: "john_doe",
 profilePicture: "profile_url"
}

sends a **JSON response** back to the frontend with the user's posts data.

likepost

```
export const likePost = async (req,res)=>{
  try {
    const likeKrneWalaUserKiId = req.id;
    const postId = req.params.id;
    const post = await Post.findById(postId);
    if (!post) return res.status(404).json({ message: 'Post not found', success: false });
    await post.updateOne({$addToSet: { likes: likeKrneWalaUserKiId } });
    await post.save();
    const user = await User.findById(likeKrneWalaUserKiId).select('username profilePicture');
    const postOwnerId = post.author.toString();
    if (postOwnerId !== likeKrneWalaUserKiId) {
      const notification = {
        type: 'like',
        userId : likeKrneWalaUserKiId,
        userDetails: user,
        postId: post._id,
        message : `${user.username} liked your post.`
      };
    }
  }
}
```

req.id comes from JWT authentication middleware (who is liking)

req.params.id comes from the URL like

```
const [liked, setLiked] = useState(post.likes.includes(user?._id) || false);
```

Adds user ID to the likes array ONLY if it's not already there

Prevents duplicates: Same user can't like the same post twice

Like HashSet in Java: No duplicate values allowed

Creates notification object to send on socketio

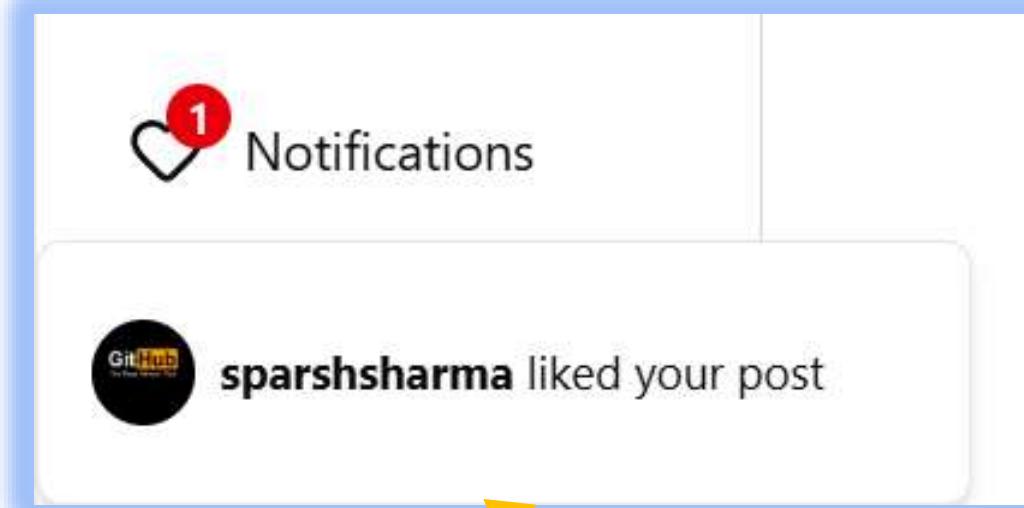
Save it to database

```
export const getReceiverSocketId = (receiverId) => userSocketMap[receiverId];

io.on('connection', (socket) =>
  const userId = socket.handshake.query.userId;
  if (userId) {
    userSocketMap[userId] = socket.id;
  }

  io.emit('getOnlineUsers', Object.keys(userSocketMap));

  socket.on('disconnect', () => {
    if (userId) {
      delete userSocketMap[userId];
    }
    io.emit('getOnlineUsers', Object.keys(userSocketMap));
  });
);
```



```
const postOwnerSocketId = getReceiverSocketId(postOwnerId);
if (postOwnerSocketId) {
  io.to(postOwnerSocketId).emit('notification', notification);
}
```

Finds the post owner's active socket connection
Sends real-time notification (like Instagram's instant notifications)

```
return res.status(200).json({ message: 'Post liked successfully', success: true });
```

Post liked successfully

dislikePost

req.id comes from JWT authentication middleware (who is liking)

req.params.id comes from the URL like

```
//the logic of both is same...bs hame push ki agah pull karna padega
export const dislikePost = async (req,res)=>{
  try{
    const likeKrneWalaUserKiId = req.id;
    const postId = req.params.id;
    const post = await Post.findById(postId);
    if (!post) return res.status(404).json({ message: 'Post not found', success: false });

    // dislike logic started
    await post.updateOne({ $pull: { likes: likeKrneWalaUserKiId } });
    await post.save();

    // socketio ko frontend k baad banege...

    return res.status(200).json({ message: 'Post disliked successfully', success: true });
  }catch(error){
    return res.status(500).json({message:'Internal server error', success:false});
  }
}
```

Removes the user's ID from the post's likes array (unlike/dislike functionality)

removing the like from post

addComment

```
export const addComment = async (req,res) =>{
```

```
try {
```

```
    const postId = req.params.id;
    const commentKroneWalaUserKiId = req.id;
```

req.params.id can be manipulated by user (URL parameter)
req.id is secure (comes from verified JWT token)

```
    const {text} = req.body; //message -- req.body se aayega
```

```
    const post = await Post.findById(postId);
```

```
    if(!text) return res.status(400).json({message:'text is required', success:false});
```

```
    const comment = await Comment.create({
        text,
        author:commentKroneWalaUserKiId,
        post:postId
    })
```

```
    await comment.populate({
        path:'author',
        select:"username profilePicture"
    });
```

Creates and saves a new comment to the database

It contains the comment from req.body

The ID of the user who wrote the comment

The ID of the post being commented on

```
const {posts} = useSelector(store=>store.post);
```

Adds comment ID to post's array

Save to database

```
    post.comments.push(comment._id);
```

```
    await post.save();
```

Getting comments of post

```
export const getCommentsOfPost = async (req,res) => {
  try {
    const postId = req.params.id;

    const comments = await Comment.find({post:postId}).populate('author', 'username profilePicture');

    if(!comments) return res.status(404).json({message:'No comments found for this post', success:false});

    return res.status(200).json({success:true,comments});
  } catch (error) {
    console.log(error);
    return res.status(500).json({message:'Internal server error', success:false});
  }
};
```

It **fetches all comments** for a given post and replaces each comment's author ID with the author's details (username and profilePicture) from the User collection.

Extracts the post ID from the URL parameters

Example URL: GET

/api/v1/post/67890/comments → postId = "67890"

Finds all comments where the **post** field matches the given **postId**

What Frontend Receives:

```
{
  "success": true,
  "comments": [
    {
      "_id": "comment123",
      "text": "Amazing photo!",
      "post": "post456",
      "author": {
        "_id": "user789",
        "username": "john_doe",
        "profilePicture": "https://cloudina
      },
      "createdAt": "2025-10-05T12:00:00Z"
    },
    ...
  ]
}
```

DeletePost

```
export const deletePost = async (req, res) => {
  try {
    const postId = req.params.id;
    const authorId = req.id;

    const post = await Post.findById(postId);
    if (!post) return res.status(404).json({ message: 'Post not found', success: false });

    const SPECIAL_USER_ID = process.env.SPECIAL_USER_ID;

    if (
      post.author.toString() !== authorId &&
      authorId !== process.env.SPECIAL_USER_ID
    ) {
      return res.status(403).json({ message: 'Forbidden', success: false });
    }

    await Post.findByIdAndDelete(postId);

    let user = await User.findById(authorId);

    if (user) {
      user.posts = user.posts.filter(id => id.toString() !== postId);
      await user.save();
    }

    await Comment.deleteMany({ post: postId });

    return res.status(200).json({
      success: true,
      message: 'Post deleted'
    });
  }
}
```

req.params.id can be manipulated by user (URL parameter)
req.id is secure (comes from verified JWT token)

Finds the post in database by ID

post.author.toString() !== authorId → checks if the person making the request is **not the post's author**.

authorId !== process.env.SPECIAL_USER_ID → checks if that person is also **not a special admin user** (whose ID is stored in environment variables).

Example:

```
// Before: user.posts = ["post1", "post2", "post3"]
// After:  user.posts = ["post1", "post3"] // post2 removed
```

It cleans the user's post list so that the deleted post ID is no longer linked to them

Delete associated comments



Post deleted

This is the bookmark post logic

```
export const bookmarkPost = async (req,res) => {
  try {
    const postId = req.params.id;
    const authorId = req.id;

    const post = await Post.findById(postId);
    if(!post) return res.status(404).json({message:'Post not found', success:false});

    const user = await User.findById(authorId);
    if(user.bookmarks.includes(post._id)){
      await user.updateOne({$pull:{bookmarks:post._id}});
      await user.save();
      return res.status(200).json({type:'unsaved', message:'Post removed from bookmark', success:true});
    }else{
      await user.updateOne({$addToSet:{bookmarks:post._id}});
      await user.save();
      return res.status(200).json({type:'saved', message:'Post bookmarked', success:true});
    }
  } catch (error) {
    console.log(error);
  }
}
```

req.params.id can be manipulated by user (URL parameter)
req.id is secure (comes from verified JWT token)

Checks if the post is **already bookmarked** by the user

Removes (\$pull) that post ID from the user's bookmarks array, then saves.

✓ Post removed from bookmark

If the post is not bookmarked

→ Adds (\$addToSet) the post ID to the bookmarks array (ensures no duplicates), then saves.

✓ Post bookmarked

error

sendMessage

```
export const sendMessage = async (req,res) => {
  try {
    const senderId = req.id;
    const receiverId = req.params.id;
    const {textMessage:message} = req.body;
```

{textMessage:message}: Destructures message text from request body and renames it to message

```
let conversation = await Conversation.findOne({
  participants:{$all:[senderId, receiverId]}
});
```

Find all conversations between sender and receiver

```
if(!conversation){
  conversation = await Conversation.create({
    participants:[senderId, receiverId]
  });
}
```

First-time chat: Creates new conversation if none exists

Participants array: Stores both user IDs
Auto-initialization: Sets up conversation infrastructure

```
const newMessage = await Message.create({
  senderId,
  receiverId,
  message
});
if(newMessage) conversation.messages.push(newMessage._id);
```

Saves message to database

Creates (and saves) a new Message document containing the sender, receiver and message text.

If message creation succeeded, push the message's _id into the conversation's messages array (linking the message to the conversation).

```
await Promise.all([conversation.save(), newMessage.save()])
```

→ Saves both document

```
const receiverSocketId = getReceiverSocketId(receiverId);  
if (receiverSocketId) {  
  
    io.to(receiverSocketId).emit('newMessage', newMessage);
```

getReceiverSocketId(): Maps user ID to their active socket connection
Online check: Only proceeds if receiver is currently online
Returns: Socket ID or null if user offline

Targets specific user's socket

```
try {  
    const sender = await User.findById(senderId)  
        .select('username profilePicture');
```

Complete message object with all data

```
const messageNotification = {  
    type: 'message',  
    userId: senderId,  
    userDetails: sender,  
    text: message,  
    conversationId: conversation._id,  
};
```

Complete message object with all data

User details, message text, conversation reference

```
} catch (e) {  
}  
    return res.status(201).json({  
        success: true, newMessage})
```

Emit a messageNotification to the receiver.

Frontend can update UI immediately

getMessage function

```
export const getMessage = async (req, res) => {  
  try {
```

```
    const senderId = req.id;  
    const receiverId = req.params.id;
```

req.params.id can be manipulated by user (URL parameter)
req.id is secure (comes from verified JWT token)

```
    const conversation = await Conversation.findOne({  
      participants: {$all: [SenderId, receiverId]}  
    }).populate('messages');
```

```
// If senderId = "507f1f77bcf86cd799439011"  
// If receiverId = "507f1f77bcf86cd799439012"  
  
// This will find a conversation where participants array contains BOTH IDs:  
// ✓ Matches: { participants: ["507f1f77bcf86cd799439011", "507f1f77bcf86cd799439012"] }  
// ✓ Matches: { participants: ["507f1f77bcf86cd799439012", "507f1f77bcf86cd799439011"] }  
// ✗ Won't match: { participants: ["507f1f77bcf86cd799439011"] } // Only one participant  
// ✗ Won't match: { participants: ["507f1f77bcf86cd799439011", "someOtherId"] } // Wrong
```

MongoDB Array Operator that matches documents where an array field contains **ALL** the specified elements
Order doesn't matter - [user1, user2] matches [user2, user1]
Must contain both - won't match if only one participant exists

```
}).populate('messages');
```

Before populate:

```
// Conversation document looks like this:  
{  
  _id: "...",  
  participants: ["user1Id", "user2Id"],  
  messages: ["msg1Id", "msg2Id", "msg3Id"] // Just IDs  
}
```

```
if(!conversation) return res.status(200)  
  .json({success:true, messages:[]});
```

```
return res.status(200)  
  .json({success:true, messages:conversation?.messages});
```

Return the messages

```
} catch (error) {  
  console.log(error);  
}
```

After populate:

```
// Conversation document looks like this:  
{  
  _id: "...",  
  participants: ["user1Id", "user2Id"],  
  messages: [ // Full message objects  
    {  
      _id: "msg1Id",  
      senderId: "user1Id",  
      receiverId: "user2Id",  
      message: "Hello!",  
      createdAt: "2024-01-01T10:00:00Z"  
    },  
    ...  
  ]  
}
```

Salman khan



Salman khan

[View profile](#)

Messages...

Send

If no messages are there ..it returns a empty array

Else throw error

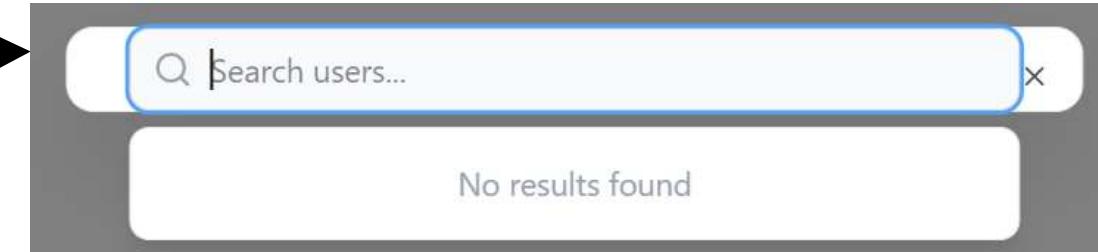
SearchUser function

```
export const searchUsers = async(req,res)=>{
  try{
    const { query } = req.query; //get search query from URL parameters
    const currentUserId = req.id; //logged in user id

    if(!query || query.trim() === ''){
      return res.status(400).json({
        message: "Search query is required",
        success: false,
      })
    }

    const users = await User.find({
      _id: { $ne: currentUserId },
      username: { $regex: query, $options: 'i' }
    }).select('-password')
      .limit(20)
      .sort({ username: 1 });

    return res.status(200).json({
      message: "Search results fetched successfully",
      success: true,
      users,
      totalResults: users.length
    })
  }
}
```



Prevents users from finding themselves in search results

\$regex: MongoDB regular expression operator for pattern matching
query: The search term entered by user (e.g., "john")
\$options: 'i': Case-insensitive flag

Exclude password field from results

Show only 20 users

Sort by username in ascending order (A-Z)

Database Search:
// MongoDB will look for:
{
 _id: { \$ne: "507f1f77bcf86cd799439011" },
 username: { \$regex: "john", \$options: 'i' }
}

seeFollowers

```
export const seeFollowers = async(req,res)=>{  
  try{  
    const userId = req.params.id;
```

```
const user = await User.findById(userId)  
.select("followers")  
.populate({ path: 'followers', select: 'username profilePicture bio' });
```

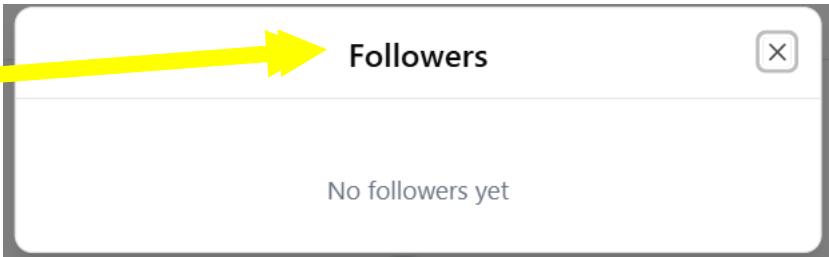
```
if(!user){  
  return res.status(404).json({  
    message:"User not found",  
    success:false,  
  })  
}  
else{  
  return res.status(200).json({  
    message:"Followers fetched successfully",  
    success:true,  
    followers: user.followers || [],  
  })  
}  
}  
catch(error){  
  console.log(error);  
  return res.status(500).json({  
    message:"Internal server error",  
    success:false,  
  })  
}
```

```
followers:[ {type:mongoose.Schema.Types.ObjectId, ref:'User'}],
```

shin chan Unfollow Message
3 posts 1 followers 1 following



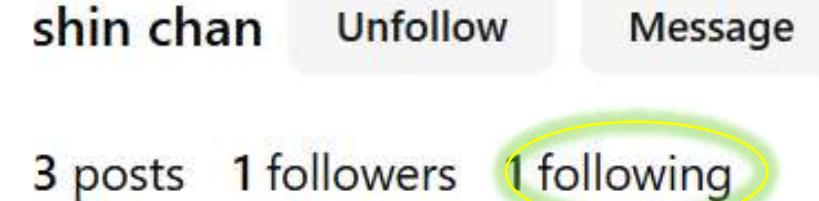
This fetches a user's **followers list**, and for each follower, you get their **username, profile picture, and bio**.



seeFollowing

```
export const seeFollowing = async(req,res)=>{
  try{
    const userId = req.params.id;
    const user = await User.findById(userId)
      .select("following")
      .populate({ path: 'following',
        | select: 'username profilePicture bio' });
    if(!user){
      return res.status(404).json({
        message:"User not found",
        success:false,
      })
    }
    return res.status(200).json({
      message:"Following fetched successfully",
      success:true,
      following: user.following || [],
    })
  catch(error){
    console.log(error);
    return res.status(500).json({
      message:"Internal server error",
      success:false,
    })
  }
}
```

It gets a user's **following** list along with each followed user's **basic info** (username, picture, bio).



following:[{type:mongoose.Schema.Types.ObjectId, ref:'User'}],

removeUser

This function is used by owner only

```
export const removeUser = async (req, res) => {
  try {
    const SPECIAL_USER_ID = process.env.SPECIAL_USER_ID;
    const currentUserId = req.id;
    const userIdToRemove = req.params.id;

    if (currentUserId !== SPECIAL_USER_ID) {
      return res.status(403)
        .json({ message: 'Forbidden', success: false });
    }

    const user = await User.findById(userIdToRemove);
    if (!user) {
      return res.status(404)
        .json({ message: 'User not found', success: false });
    }

    await Post.deleteMany({ author: userIdToRemove });
    await Comment.deleteMany({ author: userIdToRemove });
    await User.findByIdAndUpdate(userIdToRemove);

    await User.updateMany(
      { followers: userIdToRemove },
      { $pull: { followers: userIdToRemove } }
    );
    await User.updateMany(
      { following: userIdToRemove },
      { $pull: { following: userIdToRemove } }
    );
  }
}
```

owner

Check if the user is owner or not

Remove all posts create by the user

Delete all comments made by user

Finally remove the account of user

Removing user from others' followers list

Removing user from others' following list

```
await Post.updateMany(  
  { likes: userIdToRemove },  
  { $pull: { likes: userIdToRemove } }  
);
```

Finds all posts where this user's ID exists in the likes array.
\$pull removes their ID from each post's likes.

```
await Conversation.deleteMany({  
  participants: userIdToRemove  
});  
await Message.deleteMany({  
  $or: [  
    { senderId: userIdToRemove },  
    { receiverId: userIdToRemove }  
  ]  
});
```

Deletes every conversation document where this user was one of the participants.

\$or condition matches messages where the user was **either the sender or receiver**.
deleteMany removes all such messages.

```
const userPosts = await Post.find({ author: userIdToRemove }).select('_id');  
const userPostIds = userPosts.map(post => post._id);
```

Finds all posts where author matches the userIdRemove
.select('id') means only fetch the _id field of each post(no captions, images)

✓ **Result:** an array of post objects like:

```
if (userPostIds.length > 0) {  
  await User.updateMany(  
    { bookmarks: { $in: userPostIds } },  
    { $pull: { bookmarks: { $in: userPostIds } } }  
  );  
}
```

```
return res.status(200)  
.json({ message: 'User removed successfully',  
success: true });
```

Creates a simple array of post IDs:

Remove those posts from everyone's bookmarks

Users removed...

Mongodb Transaction session

All or nothing

A MongoDB Transaction Session is a way to group multiple database operations together so they either **ALL succeed** or **ALL fail** as a single unit. It's like a safety net for your data integrity.

Real Example - Your `removeUser` Function:

Without Transactions (Current Implementation):

```
// ✗ PROBLEM: If any operation fails, you get partial deletion
await Post.deleteMany({ author: userIdToRemove }); // ✓ Succeeds
await Comment.deleteMany({ author: userIdToRemove }); // ✗ Fails
await User.findByIdAndDelete(userIdToRemove); // ✗ Never executes

// Result: Posts deleted but user still exists! 🤦
```

Basic Transaction Structure:

```
import mongoose from 'mongoose';

const session = await mongoose.startSession(); // Create session
session.startTransaction(); // Begin transaction

try {
  // Your database operations here (with session)
  await User.create([{ name: "John" }], { session });
  await Post.create([{ title: "Hello" }], { session });

  await session.commitTransaction(); // Save all changes
  console.log("Transaction successful!");
}

} catch (error) {
  await session.abortTransaction(); // Undo all changes
  console.log("Transaction failed, all changes rolled back");
  throw error;
}

} finally {
  session.endSession(); // Clean up
}
```

RemoveUser with mongodb transaction session

```
export const removeUser = async (req, res) => {
  const session = await mongoose.startSession();
  session.startTransaction();
  try {
    const SPECIAL_USER_ID = process.env.SPECIAL_USER_ID;
    const currentUserId = req.id; // logged-in user's id
    const userIdToRemove = req.params.id; // user to be removed
    if (currentUserId !== SPECIAL_USER_ID) {
      await session.abortTransaction();
      return res.status(403)
        .json({ message: 'Forbidden', success: false });
    }
    const user = await User.findById(userIdToRemove).session(session);
    if (!user) {
      await session.abortTransaction();
      return res.status(404)
        .json({ message: 'User not found', success: false });
    }
    const userPosts = await Post.find({ author: userIdToRemove })
      .select('_id').session(session);
    const userPostIds = userPosts.map(post => post._id);

    await Promise.all([
      Post.deleteMany({ author: userIdToRemove }, { session }),
      Comment.deleteMany({ author: userIdToRemove }, { session }),
      User.updateMany(
        { followers: userIdToRemove },
        { $pull: { followers: userIdToRemove } },
        { session }
      ),
      User.updateMany(
        { following: userIdToRemove },
        { $pull: { following: userIdToRemove } },
        { session }
      ),
      Post.updateMany(
        { likes: userIdToRemove },
        { $pull: { likes: userIdToRemove } },
        { session }
      ),
    ]);
  }
}
```

```
  Conversation.deleteMany(
    { participants: userIdToRemove },
    { session }
  ),
  Message.deleteMany(
    {
      $or: [
        { senderId: userIdToRemove },
        { receiverId: userIdToRemove }
      ]
    },
    { session }
  )
  if (userPostIds.length > 0) {
    await User.updateMany(
      { bookmarks: { $in: userPostIds } },
      { $pull: { bookmarks: { $in: userPostIds } } },
      { session }
    );
  }
  await User.findByIdAndDelete(userIdToRemove, { session });
  await session.commitTransaction();
  return res.status(200).json({
    message: 'User and all associated data removed successfully',
    success: true
  });

} catch (error) {
  await session.abortTransaction();
  console.log('Error in removeUser:', error);
  return res.status(500).json({
    message: 'Failed to remove user. All operations have been rolled back.',
    success: false
  });
} finally {
  session.endSession();
}
```

Message.route.js

```
app.use("/api/v1/message",messageRoute);
```

backend > Routes > **JS** message.route.js > **[Θ]** default

```
1 import express from "express";
2 | import isAuthenticated from "../middlewares/isAuthenticated.js";
3 import upload from "../middlewares/multer.js";
4 import { getMessage, sendMessage } from "../controllers/message.controller.js";
5
6 const router = express.Router();
7
8 router.route('/send/:id').post(isAuthenticated, sendMessage);
9 router.route('/all/:id').get(isAuthenticated, getMessage);
10
11 export default router;
```

Post routes

```
import express from "express";
import isAuthenticated from "../middlewares/isAuthenticated.js";
import upload from "../middlewares/multer.js";
import { addComment, addNewPost, bookmarkPost, deletePost, dislikePost, getAllPosts,
    getCommentsOfPost, getUserPost, likePost } from "../controllers/post.controller.js";

const router = express.Router();

router.route("/addpost").post(isAuthenticated, upload.single('image'), addNewPost);
//authenticated user hi post kar sakta hai --- single image hi upload kar sakte hai

router.route("/all").get(isAuthenticated, getAllPosts);

router.route("/userpost/all").get(isAuthenticated, getUserPost);

router.route("/:id/like").get(isAuthenticated, likePost);

router.route("/:id/dislike").get(isAuthenticated, dislikePost);

router.route("/:id/comment").post(isAuthenticated, addComment);

router.route("/:id/comment/all").post(isAuthenticated, getCommentsOfPost);

router.route("/delete/:id").delete(isAuthenticated, deletePost);

router.route("/:id/bookmark").get(isAuthenticated, bookmarkPost);

export default router;
```

User routes

```
import express from "express";
import { editProfile, followOrUnfollowUser, getProfile, getSuggestedUsers, login,
    logout, register, searchUsers, seeFollowers,
    seeFollowing, removeUser } from "../controllers/user.controller.js";

import isAuthenticated from "../middlewares/isAuthenticated.js";

import upload from "../middlewares/multer.js";

const router = express.Router();

router.route("/register").post(register);

router.route("/login").post(login);

router.route("/logout").get(logout);

router.route("/:id/profile").get(isAuthenticated, getProfile);

router.route("/profile/edit").post(isAuthenticated, upload.single("profilePhoto"), editProfile);

router.route("/suggested").get(isAuthenticated, getSuggestedUsers);

router.route("/:id/followers").get(isAuthenticated, seeFollowers);
router.route("/:id/following").get(isAuthenticated, seeFollowing);

router.route("/search").get(isAuthenticated, searchUsers);

router.route("/followorunfollow/:id").post(isAuthenticated, followOrUnfollowUser);

router.delete('/remove/:id', isAuthenticated, removeUser);
```

Then we will populate to get the details of post

```
//we want information about the post on the basis of its id
const populatedPosts = await Promise.all(
  user.posts.map( async (postId) => {
    const post = await Post.findById(postId);
    if(post.author.equals(user._id)){
      return post;
    }
    return null;
  })
)
//post array ke andar user ki saari post hai
```