**Backend for User**

**Controller**

**Middlewares**

**Models**

**Routes**

**Controller :** A controller refers to a component or module that handles the logic and flow of an application.controllers in the backend are responsible for processing incoming requests, coordinating the necessary actions, and preparing the response to be sent back to the client

1. **usercontroller.js**

| **require('dotenv').config();**  **const bcrypt = require('bcrypt');**  **const jwt = require('jsonwebtoken');**  **const User = require('../models/User');**  **const getUserById = async (req, res) => {**  **const userId = req.params.id; // Assuming the user ID is passed as a parameter in the request**  **try {**  **// Implement your user retrieval logic here, e.g., interacting with a database**  **const user = await user.findById(userId); // Assuming you have a "User" model or collection**  **if (!user) {**  **// If the user is not found, return a 404 response**  **return res.status(404).json({ error: 'User not found' });**  **}**  **// If the user is found, return the user data**  **return res.json(user);**  **} catch (error) {**  **// If there is an error, return an error response**  **return res.status(500).json({ error: 'Internal server error' });**  **}**  **};**  **const updateUserById = async (req, res) => {**  **const userId = req.params.id; // Assuming the user ID is passed as a parameter in the request**  **const updateData = req.body; // Assuming the update data is provided in the request body**  **try {**  **// Implement your user update logic here, e.g., interacting with a database**  **const updatedUser = await User.findByIdAndUpdate(userId, updateData, { new: true });**  **if (!updatedUser) {**  **// If the user is not found, return a 404 response**  **return res.status(404).json({ error: 'User not found' });**  **}**  **// If the user is successfully updated, return the updated user data**  **return res.json(updatedUser);**  **} catch (error) {**  **// If there is an error, return an error response**  **return res.status(500).json({ error: 'Internal server error' });**  **}**  **};**  **const searchUsers = async (req, res) => {**  **try {**  **const { age, gender } = req.query;**  **const dateOfBirth = new Date();**  **dateOfBirth.setFullYear(dateOfBirth.getFullYear() - age);**    **// Add additional search criteria if needed**  **const result = await User.find({ $or: [{ dateOfBirth: { $lte: dateOfBirth } }, { gender: gender }] });**  **res.status(200).json(result);**  **} catch (error) {**  **res.status(500).json({ message: 'Server Error' });**  **}**  **};**  **module.exports = { getUserById, updateUserById, searchUsers };** |
| --- |

**require('dotenv').config() :** This line imports and executes the dotenv package, which allows you to load environment variables from a .env file into the Node.js runtime environment.

**const bcrypt = require('bcrypt'):** This line imports the bcrypt package, which provides functions for hashing and comparing passwords. Bcrypt is a popular library for securely storing passwords by applying a one-way hash function.

**const jwt = require('jsonwebtoken') :** This line imports the jsonwebtoken package, which enables the creation and verification of JSON Web Tokens (JWTs). JWTs are used for authentication and authorization purposes in web applications.

**const User = require('../models/User'):** This line imports the user model from the '../models/User' file.

**getUserById**: Retrieves a user by their ID from the database and returns their data.

**updateUserById**: Updates a user's data in the database based on their ID using the provided update data and returns the updated user.

**searchUsers**: Searches for users in the database based on age and gender criteria provided in the request query parameters and returns the matching results.

**module.exports = { getUserById, updateUserById, searchUsers }:** This allows other modules or files to import and use these functions by requiring the module that exports them.

**2 .profileController.js**

| **const User = require('../models/User');**  **exports.getProfile = async (req, res) => {**  **try {**  **const result = await User.find();**  **res.status(200).json(result);**  **} catch (error) {**  **console.error(error);**  **res.status(500).json({ message: 'Internal server error' });**  **}**  **}**  **exports.getSingleProfile = async (req, res) => {**  **try {**  **const { userId } = req.decoded;**  **const user = await User.findById(userId);**  **if (!user) return res.status(404).json({ message: 'User not found' });**    **res.status(200).json(user);**  **} catch (error) {**  **res.status(500).json({ message: 'Internal server error' });**  **}**  **}**  **// Update user profile**  **exports.updateProfile = async (req, res) => {**  **const { bio, location, website } = req.body;**  **const userId = req.userId;**  **try {**  **// Find the user by their ID**  **const user = await User.findById(userId);**  **if (!user) {**  **return res.status(404).json({ message: 'User not found' });**  **}**  **// Update the profile fields**  **user.bio = bio || user.bio;**  **user.location = location || user.location;**  **user.website = website || user.website;**  **// Save the updated user document**  **await user.save();**  **res.status(200).json({ message: 'Profile updated successfully' });**  **} catch (error) {**  **console.error(error);**  **res.status(500).json({ message: 'Internal server error' });**  **}**  **};** |
| --- |

**const User = require('../models/User'):**This line imports the user model from the '../models/User' file.

**getProfile**: Retrieves all user profiles from the database and returns them as a JSON response with a status code of 200. If an error occurs during the retrieval process, it returns a JSON response with a status code of 500.

**getSingleProfile**: Retrieves a single user profile from the database based on the authenticated user's ID. It uses the req.decoded object, which likely contains the decoded information from a JWT, to extract the userId. If the user is not found, it returns a JSON response with a status code of 404. Otherwise, it returns the user profile as a JSON response with a status code of 200. If an error occurs, it returns a JSON response with a status code of 500.

**updateProfile**: Updates a user's profile fields (bio, location, and website) based on the data provided in the request body. It first finds the user by their ID using User.findById(). If the user is not found, it returns a JSON response with a status code of 404.

**3. Matchcontroller.js:**

| **const User = require('../models/User');**  **exports.sendConnectionRequest = async (req, res) => {**  **try {**  **const { userId, connectionId } = req.body;**  **// Add the connectionId to the user's connections array**  **await User.findByIdAndUpdate(userId, {**  **$addToSet: { connections: connectionId },**  **});**  **res.json({ message: 'Connection request sent successfully' });**  **} catch (error) {**  **console.error(error);**  **res.status(500).json({ message: 'Server Error' });**  **}**  **};** |
| --- |

**const User = require('../models/User'):**This line imports the user model from the '../models/User' file.

**sendConnectionRequest:** Handles the logic for sending a connection request between users by updating the connections array of the user initiating the request with the provided connection ID.

**getUserById**: Retrieves a user profile from the database based on the provided user ID.

**updateUserById**: Updates a user's profile in the database with the provided update data based on the user ID.

**searchUsers**: Searches for users in the database based on the provided age and gender criteria.

**4. authController.js:**

| **const User = require('../models/User');**  **const bcrypt = require('bcrypt');**  **const jwt = require('jsonwebtoken');**  **// Register a new user**  **exports.register = async (req, res) => {**  **const { name, email, password, dateOfBirth, gender, location } = req.body;**  **try {**  **// Check if the user already exists**  **const existingUser = await User.findOne({ email });**  **if (existingUser) {**  **return res.status(400).json({ message: 'Email already registered' });**  **}**  **// Hash the password**  **const hashedPassword = await bcrypt.hash(password, 10);**  **// Create a new user document**  **const newUser = new User({**  **name,**  **email,**  **password: hashedPassword,**  **dateOfBirth,**  **gender,**  **location,**  **});**  **// Save the user document to the database**  **const result = await newUser.save();**  **res.status(200).json({ message: 'User registered successfully' });**  **} catch (error) {**  **console.error(error);**  **res.status(500).json({ message: 'Internal server error' });**  **}**  **};**  **// User login**  **exports.login = async (req, res) => {**  **const { email, password } = req.body;**  **try {**  **// Check if the user exists in the database**  **const user = await User.findOne({ email });**  **if (!user) {**  **return res.status(404).json({ message: 'User not found' });**  **}**  **// Compare the provided password with the stored hashed password**  **const isPasswordMatch = await bcrypt.compare(password, user.password);**  **if (!isPasswordMatch) {**  **return res.status(401).json({ message: 'Invalid credentials' });**  **}**  **// Create and sign a JSON Web Token (JWT)**  **const token = jwt.sign({ userId: user.\_id }, 'user\_token');**  **// Send the token in the response**  **res.status(200).json({ token });**  **} catch (error) {**  **console.error(error);**  **res.status(500).json({ message: 'Internal server error' });**  **}**  **};**  **module.exports = exports;** |
| --- |

**require('../models/User')**: This line requires the User model, which presumably defines the schema and methods for interacting with the user collection in MongoDB.

**bcrypt**: This package is used for hashing the user's password for security purposes.

**jsonwebtoken**: This package is used for generating JSON Web Tokens (JWTs) for user authentication and authorization.

**Register :** function checks if a user with the provided email already exists. If not, it hashes the password, creates a new user document, and saves it to the database.

**Login:** function checks if a user with the provided email exists. If so, it compares the provided password with the hashed password stored in the user document. If the passwords match, it generates a JSON Web Token and sends it in the response.

Both functions handle errors and return appropriate HTTP status codes and error messages.

**Middleware**

Middleware is a concept commonly used in web development to process and modify incoming requests before they reach the actual route handlers. In the context of the user registration and login code you provided, middleware can be used to perform additional checks or operations on the requests.

1. **authmiddleware**.**js**

| const jwt = require('jsonwebtoken');  const authMiddleware = (req, res, next) => {  // Get the token from the request headers  const token = req.headers.authorization;  // Check if a token exists  if (!token) {  return res.status(401).json({ message: 'No token, authorization denied' });  }  try {  // Verify and decode the token  const decoded = jwt.verify(token, 'your\_secret\_key', function (err, decoded) {  if (err) return res.status(403).json({ message: "Invlid token" });    // Attach the decoded user ID to the request object  req.decoded = decoded;  next();  });  } catch (error) {  console.error(error);  res.status(401).json({ message: 'Invalid token' });  }  };  module.exports = authMiddleware; |
| --- |

**const jwt** = require('jsonwebtoken'): This line imports the jsonwebtoken package, which is used for verifying and decoding JWTs.

**const authMiddleware** = (req, res, next) => { ... }: This defines the authentication middleware function that takes in the req, res, and next parameters. The next parameter is a function that allows the middleware to pass control to the next middleware or route handler.

**const token** = req.headers.authorization;: This line retrieves the token from the Authorization header of the incoming request.

**if (!token) { ... }**: This condition checks if a token exists. If not, it returns a 401 (Unauthorized) response with an error message.

**jwt.verify(token, 'your\_secret\_key', function (err, decoded) { ... }**: This line verifies and decodes the token using the provided secret key. The jwt.verify method takes the token and secret key as arguments, and it also takes a callback function that is executed after the verification is complete. The callback function receives two parameters: err and decoded. If an error occurs during verification, such as an expired or invalid token, it returns a 403 (Forbidden) response with an error message. Otherwise, it attaches the decoded user ID to the req object and calls next() to pass control to the next middleware or route handler.

**req.decoded = decoded;**: This line attaches the decoded user ID to the req object, making it accessible in subsequent middleware or route handlers.

1. **errorhandler.js:**

const errorHandler = (err, req, res, next) => {

console.error(err);

res.status(500).json({ message: 'Internal server error' });

};

module.exports = errorHandler;

**errorHandler**: This function is designed to handle errors that occur during the execution of route handlers or middleware functions. It takes four parameters: err (the error object), req (the request object), res (the response object), and next (the next middleware or route handler function).

**console.error(err)**: This line logs the error to the console for debugging and error tracking purposes.

**res.status(500).json({ message: 'Internal server error' })**: This line sends a 500 (Internal Server Error) response with a JSON object containing an error message.

**module.exports = errorHandler;**: This line exports the errorHandler function as a module, making it available for use in other parts of the application.

By using this error handling middleware,we can catch and handle errors that occur during the execution of route handlers or middleware functions. It provides a centralized location to handle and respond to errors, improving the overall error handling and user experience of our application.

**Models :** Models represent the data and logic of an application. They define the structure and behavior of the data entities that the application interacts with, such as database tables or external APIs.

1. **user.js**

| **const mongoose = require('mongoose');**  **const userSchema = new mongoose.Schema(**  **{**  **name: {**  **type: String,**  **required: true,**  **},**  **email: {**  **type: String,**  **required: true,**  **unique: true,**  **},**  **password: {**  **type: String,**  **required: true,**  **},**  **isAdmin: {**  **type: Boolean,**  **default: false,**  **},**  **dateOfBirth: {**  **type: Date,**  **required: true,**  **},**  **gender: {**  **type: String,**  **enum: ['male', 'female'],**  **required: true,**  **},**  **bio: {**  **type: String,**  **default: '',**  **},**  **location: {**  **type: String,**  **default: '',**  **},**    **// Add fields for match functionality**  **age: {**  **type: Number**  **},**  **gender:**  **{ type: String**  **},**  **// Other match-related fields**  **connections:**  **[{ type: mongoose.Schema.Types.ObjectId,**  **ref: 'User' ,**  **}],**  **},**    **{**  **timestamps: true,**  **}**      **);**  **const User = mongoose.model('User', userSchema);**  **module.exports = User;** |
| --- |

**const mongoose = require('mongoose');**: This line imports the Mongoose library, allowing you to interact with MongoDB.

**const userSchema = new mongoose.Schema({ ... });**: This code defines the schema or structure of the User model using mongoose.Schema. It specifies the fields or properties of a user document and their respective data types and validation rules.

Inside the userSchema object, the following fields are defined:

**name**: A required string field for the user's name.

**email**: A required string field for the user's email, which must be unique.

**password**: A required string field for the user's password.

**isAdmin**: A boolean field indicating whether the user is an admin. It has a default value of false.

**dateOfBirth**: A required date field for the user's date of birth.

**gender**: A required string field for the user's gender. It can only be one of the values 'male' or 'female'.

**bio**: A string field for the user's bio, with a default value of an empty string.

**location**: A string field for the user's location, with a default value of an empty string.

age and gender: Additional fields for match functionality, possibly used for matching users based on age and gender.

**{ timestamps: true }**: This option adds createdAt and updatedAt fields to the user documents, automatically managing the timestamps when the document is created or updated.

**const User = mongoose.model('User', userSchema);**: This line creates the User model using the mongoose.model method. It takes two arguments: the model name 'User' and the schema (userSchema) that defines the structure of the user documents.

**module.exports = User;**: This line exports the User model to make it available for use in other parts of the application.

Overall, the code defines a Mongoose model for a user document, specifying the fields, their data types, and validation rules. It also includes additional fields for match functionality and manages timestamps for document creation and updates.

**Routes**

are responsible for handling incoming HTTP requests and returning appropriate responses. They define the endpoints or URLs that clients can access to interact with the application.

1.**userRoutes.js**

| **const express = require('express');**  **const router = express.Router();**  **const authMiddleware = require('../middlewares/authMiddleware');**  **const { register, login, } = require('../controllers/authController');**  **const { getUserById, updateUserById, searchUsers } = require('../controllers/userController');**    **// User registration route**  **router.post('/register', register);**  **// User login route**  **router.post('/login', login);**  **router.get('/users/:id',authMiddleware, getUserById);**  **router.put('/users/:id', authMiddleware, updateUserById);**  **router.get('/search', searchUsers)**  **module.exports = router;** |
| --- |

**const express** = require('express');: This line imports the Express.js framework.

**const router** = express.Router();: This creates an instance of the Express router, which is used to define routes.

**const authMiddleware** = require('../middlewares/authMiddleware');: This imports the authMiddleware module, which is a middleware function for protecting routes that require authentication.

**const { register, login } = require('../controllers/authController')**;: This imports the register and login functions from the authController module

**const { getUserById, updateUserById, searchUsers } = require('../controllers/userController');**: This imports the getUserById, updateUserById, and searchUsers functions from the userController module.

**router.post('/register', register);**: This defines a route for user registration. It listens for POST requests to the '/register' endpoint and delegates the request to the register function from the authController.

**router.post('/login', login);**: This defines a route for user login. It listens for POST requests to the '/login' endpoint and delegates the request to the login function from the authController.

**router.get('/users/:id', authMiddleware, getUserById);**: This defines a route for retrieving user details by ID. It listens for GET requests to the '/users/:id' endpoint, applies the authMiddleware to protect the route, and delegates the request to the getUserById function from the userController.

**router.put('/users/:id', authMiddleware, updateUserById);**: This defines a route for updating user details by ID. It listens for PUT requests to the '/users/:id' endpoint, applies the authMiddleware to protect the route, and delegates the request to the updateUserById function from the userController.

**router.get('/search', searchUsers);**: This defines a route for searching users. It listens for GET requests to the '/search' endpoint and delegates the request to the searchUsers function from the userController.

**module.exports = router;**: This exports the router instance, making it available to be used by the main application file or other modules.

By defining these routes, you establish the endpoints that clients can access to perform user registration, login, retrieving user details, updating user details, and searching for users. The routes are associated with specific HTTP methods and URLs, and they delegate the requests to the respective controller functions for processing. The authMiddleware is used to protect certain routes that require authentication.

1. **searchRoutes.js:**

| **const express = require('express');**  **const router = express.Router();**  **const { searchMatches } = require('../controllers/matchController');**  **router.get('/query', searchMatches);**  **module.exports = router;** |
| --- |

**const express = require('express');**: This line imports the Express.js framework.

**const router = express.Router()**;: This creates an instance of the Express router, which is used to define routes.

**const { searchMatches } = require('../controllers/matchController');**: This imports the searchMatches function from the matchController module.

**router.get('/query', searchMatches);:** This defines a route for searching matches. It listens for GET requests to the '/query' endpoint and delegates the request to the searchMatches function from the matchController.

**module.exports = router;**: This exports the router instance, making it available to be used by the main application file or other modules.

By defining this route, you establish an endpoint that clients can access to perform a search for matches. The route listens for a GET request to the '/query' endpoint and delegates the request to the searchMatches function from the matchController. The searchMatches function is responsible for processing the request and returning the appropriate response.

**3.profileRoutes.js**

| const express = require('express');  const { updateProfile, getProfile, getSingleProfile } = require('../controllers/profileController');  const authMiddleware = require('../middlewares/authMiddleware');  const router = express.Router();  // Update user profile route  router.get('/', getProfile);  router.get('/single', authMiddleware, getSingleProfile);  router.put('/profile', authMiddleware, updateProfile);  module.exports = router; |
| --- |

**const express = require('express'); :** express: The Express.js framework module.

**const { updateProfile, getProfile, getSingleProfile } = require('../controllers/profileController');:** updateProfile, getProfile, and getSingleProfile: Controller functions responsible for handling profile-related operations.

**const authMiddleware = require('../middlewares/authMiddleware');:** authMiddleware: Middleware function responsible for authentication.

**const router = express.Router(); :** Create a new router instance.

The **get()** method creates a route for retrieving a user profile when the base path ('/') is accessed. It uses the getProfile controller function to handle the request.

The **get()** method creates a route for retrieving a single user profile when the '/single' path is accessed. It uses the authMiddleware middleware function to ensure authentication before accessing the profile. The getSingleProfile controller function handles the request..

The **put()** method creates a route for updating a user profile when the '/profile' path is accessed. It also uses the authMiddleware middleware function to authenticate the request. The updateProfile controller function handles the request.

**module.exports = router;** This allows the router to be imported and used in other parts of the application.

**4.authRoutes.js**

| **const express = require('express');**  **const { register, login } = require('../controllers/authController');**  **const router = express.Router();**  **router.post('/register', register);**  **router.post('/login', login);**  **module.exports = router;** |
| --- |

**const express = require('express');:** express: The Express.js framework module.

**const { register, login } = require('../controllers/authController'); :** register and login: Controller functions responsible for handling user registration and login operations.

**router.post('/register', register);:** The post() method creates a route for user registration when the '/register' path is accessed. It uses the register controller function to handle the request.

**router.post('/login', login);** : The post() method creates a route for user login when the '/login' path is accessed. It uses the login controller function to handle the request.

**Genereatetoken.js**

| **const jwt = require('jsonwebtoken');**  **dotenv.config();**  **const { JWT\_SECRET } = require('../config');**  **const generateToken = (userId) => {**  **return jwt.sign({ userId }, JWT\_SECRET, { expiresIn: '1h' });**  **};**  **module.exports = generateToken;** |
| --- |

**jsonwebtoken**: A library for generating and verifying JSON Web Tokens.

**dotenv**: A module for loading environment variables from a .env file.

**JWT\_SECRET:** A constant variable that holds the JWT secret value, likely stored in a configuration file.

generateToken function uses the **jwt.sign()** method to generate a JWT based on the provided userId. It signs the token using the JWT\_SECRET value and sets an expiration time of 1 hour (expiresIn: '1h').

The { userId } payload in the jwt.sign() method creates a JWT payload containing the userId value.

**app.js**

| **const express = require('express');**  **require('dotenv').config();**  **const cors = require('cors');**  **// import the required routes and middlewares**  **const userRoutes = require('./routes/userRoutes');**  **const profileRoutes = require('./routes/profileRoutes');**  **const errorHandler = require('./middlewares/errorHandler');**  **const authRoutes = require('./routes/authRoutes');**  **const adminRoutes = require('./routes/adminRoutes');**  **const searchRoutes = require('./routes/searchRoutes')**  **const app = express();**  **// Middleware**  **app.use(cors());**  **app.use(express.json());**  **app.use(errorHandler);**  **app.use('/api', express.static("public"));**  **// Routes**  **app.use('/api/users', userRoutes);**  **app.use('/api/profile', profileRoutes);**  **app.use('/api/auth', authRoutes);**  **app.use('/api/admin', adminRoutes);**  **module.exports = app;** |
| --- |

**express**: The Express.js framework module.

**dotenv**: A module for loading environment variables from a .env file.

**cors**: A middleware for enabling Cross-Origin Resource Sharing.

**app.use(cors())**: Enable CORS to allow cross-origin requests.

**app.use(express.json())**: Parse JSON request bodies.

**app.use(errorHandler)**: Use the errorHandler middleware to handle errors.

**app.use('/api', express.static("public"))**: Serve static files from the "public" directory for requests starting with "/api".

**app.use('/api/users', userRoutes):** Mount the userRoutes router to handle requests starting with "/api/users".

**app.use('/api/profile', profileRoutes):** Mount the profileRoutes router to handle requests starting with "/api/profile".

**app.use('/api/auth', authRoutes):** Mount the authRoutes router to handle requests starting with "/api/auth".

**app.use('/api/admin', adminRoutes):** Mount the adminRoutes router to handle requests starting with "/api/admin".

**server.js**

| **const express = require('express');**  **const mongoose = require('mongoose');**  **const authMiddleware = require('./middlewares/authMiddleware');**  **const cors = require('cors');**  **require('dotenv').config();**  **const app = express();**  **// Middleware**  **app.use(express.json());**  **app.use(cors())**  **// Routes**  **app.get('/public', authMiddleware, (req, res) => {**  **res.send('API is running');**  **});**  **app.post('/register', (req, res) => {**  **// Handle the registration logic here**  **});**  **// Authentication routes**  **app.use('/api/auth', require('./routes/authRoutes'));**  **// User routes**  **app.use('/api/profile', require('./routes/profileRoutes'));**  **app.use('/api/users', require('./routes/userRoutes'));**  **app.use('/api/admin', require('./routes/adminRoutes'));**  **// Connect to the database**  **const MONGODB\_URI = process.env.MONGODB\_URI;**  **const PORT = process.env.PORT || 3001;**  **mongoose.connect(MONGODB\_URI)**  **.then(() => {**  **console.log('Connected to MongoDB');**  **// Start the server**  **app.listen(PORT, () => {**  **console.log(`Server is running on port ${PORT}`);**  **});**  **})**  **.catch((error) => {**  **console.error('Error connecting to MongoDB:', error);**  **process.exit(1); // Terminate the server in case of connection error**  **});**  **module.exports = app;** |
| --- |

**express:** The Express.js framework module.

**mongoose**: A MongoDB object modeling tool.

**authMiddleware**: A custom middleware for authentication.

**cors**: A middleware for enabling Cross-Origin Resource Sharing.

**dotenv**: A module for loading environment variables from a .env file.

**app.use(express.json())**: Parse JSON request bodies.

**app.use(cors())**: Enable CORS to allow cross-origin requests.

**app.get('/public', authMiddleware, (req, res) => {...}):** Defines a GET route for '/public' that requires authentication middleware (authMiddleware) and sends the response 'API is running'.

**app.post('/register', (req, res) => {...}):** Defines a POST route for '/register' where you can handle the registration logic.

**app.use('/api/auth', require('./routes/authRoutes'))**: Mounts the authentication routes from the './routes/authRoutes' module.

**app.use('/api/profile', require('./routes/profileRoutes'))**: Mounts the user profile routes from the './routes/profileRoutes' module.

**app.use('/api/users', require('./routes/userRoutes'))**: Mounts the user routes from the './routes/userRoutes' module.

**app.use('/api/admin', require('./routes/adminRoutes'))**: Mounts the admin routes from the './routes/adminRoutes' module.

**MONGODB\_URI**: The MongoDB connection URI specified in the environment variables.

**PORT:** The port number on which the server will listen.

**mongoose.connect(MONGODB\_URI):** Connects to the MongoDB database using the specified URI.

Once the database connection is established, the server is started and listens on the specified port.