S erver.js

NM,./

**01. require("express")**

* Loads the **Express** framework.
* Express is used to build web servers and APIs easily in Node.js.

**2. require("mongoose")**

* Loads **Mongoose**, which is an ODM (Object Data Modeling) library for MongoDB.
* It helps you connect to MongoDB, define schemas, and interact with the database using JavaScript objects instead of raw queries.

**3. require("cors")**

* Loads **CORS (Cross-Origin Resource Sharing)** middleware.
* Used to allow your API to be accessed from different domains (for example, if your frontend is on http://localhost:3000 and your backend on http://localhost:5000).

**4. require("dotenv").config()**

* Loads environment variables from a .env file into process.env.
* For example, process.env.MONGO\_URI and process.env.PORT are read from .env.

**5. const nn K;OLKKKKKKKKKKKKKKKKImkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkKKKKMKJIK,contactRoutes = require("./routes/contactRoutes")**

* Imports your **contactRoutes** file where all the contact-related API endpoints are defined (like POST /contact, GET /contact, etc.).

**6. const app = express()**

* Creates an Express application object (app) that you’ll use to define middleware and routes.

**Middleware**

Middleware functions run **before** your routes handle requests.

**7. app.use(cors())**

* Enables **CORS** for all requests (so frontend apps can talk to this API).

**8. app.use(express.json())**

* Parses **incoming JSON data** in request bodies.
* Example: if a frontend sends { "name": "Siji" }, you can access it as req.body.name.

**9. app.use(express.urlencoded({ extended: true }))**

* Parses data sent via **HTML forms** (application/x-www-form-urlencoded).
* extended: true allows for nested objects in form data.

**Routes**

**10. app.use("/api/contact", contactRoutes)**

* Mounts all contact-related routes under /api/contact.
* Example:
  + If contactRoutes has router.post("/"), it becomes POST /api/contact.

**Database + Server Start**

**11. mongoose.connect(process.env.MONGO\_URI)**

* Connects your app to MongoDB using the connection string from .env.

**12. .then(() => { app.listen(...) })**

* Runs only if DB connection succeeds.
* Starts the Express server on the port from .env (e.g., 5000).

**13. .catch((err) => console.error("DB Connection Error:", err))**

* If database connection fails, logs the error.

✅ In short:

* **Express** → Handles server & routes
* **Mongoose** → Connects to MongoDB
* **CORS** → Allows cross-domain requests
* **dotenv** → Loads environment variables
* **Middleware (express.json, express.urlencoded)** → Parses request data
* **Routes** → Defines API endpoints
* **Mongoose connect + app.listen** → Connect DB & start server

API request life cycle

API:

**🔄 Step-by-Step Flow**

**1. Client sends a request**

Example:  
Your frontend form sends a request to:

POST http://localhost:5000/api/contact

Body: { "name": "Siji", "email": "siji@example.com", "message": "Hello" }

**2. Express receives the request**

* Since your app is created with const app = express(), Express is now listening for requests.
* It checks middleware first **before** hitting your routes.

**3. Middleware runs**

* **cors()** → Allows this request if it’s from another domain (e.g., frontend http://localhost:3000).
* **express.json()** → Reads the raw JSON body and converts it into req.body.
  + Now you can do req.body.name, req.body.email, etc.
* **express.urlencoded({ extended: true })** → If it was form-data instead of JSON, this would handle it.

**4. Routes are checked**

* You wrote:
* app.use("/api/contact", contactRoutes);
* So Express looks inside contactRoutes.
* If contactRoutes has:
* router.post("/", createContact);

Then POST /api/contact will trigger the createContact function.

**5. Route Controller executes**

* Inside your controller (createContact), you might have something like:
* const Contact = require("../models/Contact");
* const createContact = async (req, res) => {
* try {
* const newContact = new Contact(req.body);
* await newContact.save();
* res.status(201).json({ message: "Contact saved", data: newContact });
* } catch (err) {
* res.status(500).json({ error: err.message });
* }
* };
* This function:
  1. Takes request data (req.body)
  2. Creates a new **Mongoose model object**
  3. Saves it into MongoDB
  4. Responds back to client

**6. Mongoose talks to MongoDB**

* new Contact(req.body) creates a document object based on the schema you defined.
* .save() inserts it into your MongoDB collection.

**7. Response sent to client**

* If success → API returns 201 Created with JSON:
* {
* "message": "Contact saved",
* "data": {
* "\_id": "65128ad9f3...",
* "name": "Siji",
* "email": "siji@example.com",
* "message": "Hello"
* }
* }
* If error (like DB not connected) → API returns 500 with error details.

✅ **Summary of Request Lifecycle**

1. Request comes in →
2. Middleware parses data + checks CORS →
3. Routes decide which controller runs →
4. Controller executes →
5. Mongoose stores/retrieves data →
6. Response goes back to client

🖥️ Client (Frontend / Postman)

|

| POST /api/contact

| Body: { name, email, message }

v

🌐 Express Server (app.js)

|

|-- Middleware

| • cors() → allows request

| • express.json() → parse JSON

| • express.urlencoded() → parse form data

|

|-- Routes

| /api/contact --> contactRoutes.js

|

|-- Controller (e.g., createContact)

| • Receives req.body

| • Creates new Contact model

| • Calls Contact.save()

|

v

📦 Mongoose (ODM Layer)

|

v

🗄️ MongoDB Database

• Insert document into "contacts" collection

|

v

🌐 Express sends Response

| Success (201):

| { "message": "Contact saved", "data": {...} }

|

| Error (500):

| { "error": "Something went wrong" }

v

🖥️ Client Receives Response

**Contact.js**

**Mongoose model** for your Contact. Let’s break it down step by step:

**1. const mongoose = require("mongoose");**

* Imports **Mongoose**, the ODM library for MongoDB.
* You’ll use it to define schemas and interact with collections.

**2. const contactSchema = new mongoose.Schema({...}, { timestamps: true });**

This line defines a **schema** (blueprint) for your Contact documents.

**Inside the schema:**

{

name: { type: String, required: true },

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

message: { type: String, required: true },

}

* **name** → Must be a string, required (document won’t save if missing).
* **email** → Must be a string, required.
* **message** → Must be a string, required.

So every Contact document **must** have all 3 fields.

**{ timestamps: true }**

* Mongoose automatically adds:
  + createdAt → when the document was first created
  + updatedAt → when the document was last modified

Example:

{

"\_id": "65128ad9f3...",

"name": "Siji",

"email": "siji@example.com",

"message": "Hello",

"createdAt": "2025-09-25T10:30:00.123Z",

"updatedAt": "2025-09-25T10:30:00.123Z"

}

**3. module.exports = mongoose.model("Contact", contactSchema);**

* Creates a **Mongoose model** called Contact using your schema.
* A model is like a **class** for documents.
* This tells Mongoose:
  + Collection name: contacts (Mongoose auto-pluralizes Contact)
  + Structure: follows contactSchema

You’ll now be able to use this model in controllers:

const Contact = require("../models/Contact");

// Save new contact

const newContact = new Contact({

name: "Siji",

email: "siji@example.com",

message: "Hello!"

});

await newContact.save();

✅ **In short:**

* contactSchema → Blueprint for how a contact document should look.
* timestamps: true → Automatically manages createdAt & updatedAt.
* mongoose.model("Contact", contactSchema) → Creates the actual model to interact with MongoDB.
*  **Schema** → Defines how data should look.
*  **Model** → Lets you interact with the DB.
*  **Controller** → Business logic (save, fetch, delete, etc.).
*  **Route** → Connects URLs to controllers.
*  **App.js** → Starts everything + mounts routes.

ContactRoutes.js

it connects **URLs** to your **controller functions**. Let’s break it down:

**1. const express = require("express");**

* Loads **Express** so you can use its router system.

**2. const router = express.Router();**

* Creates a new **router object**.
* A router is like a mini Express app, but only for a group of routes.
* Later, in app.js, you mounted it with:
* app.use("/api/contact", contactRoutes);

→ So all these routes will start with /api/contact.

**3. const { createContact, getContacts, updateContact, deleteContact, getContactById } = require("../controllers/contactController");**

* Imports your controller functions.
* Each function has the **logic** for handling requests (save, fetch, update, delete).

**🔄 Routes**

**(a) Create a Contact**

router.post("/", createContact);

* **Method:** POST
* **URL:** /api/contact/
* **Controller:** createContact
* **Purpose:** Save a new contact form submission.

Example:

POST /api/contact

Body: { "name": "Siji", "email": "siji@example.com", "message": "Hello" }

**(b) Get All Contacts**

router.get("/", getContacts);

* **Method:** GET
* **URL:** /api/contact/
* **Controller:** getContacts
* **Purpose:** Fetch all submitted forms from DB.

**(c) Update a Contact**

router.put("/:id", updateContact);

* **Method:** PUT
* **URL:** /api/contact/:id
* **Controller:** updateContact
* **Purpose:** Update a specific contact by its MongoDB \_id.

Example:

PUT /api/contact/65128ad9f3...

Body: { "message": "Updated message" }

**(d) Delete a Contact**

router.delete("/:id", deleteContact);

* **Method:** DELETE
* **URL:** /api/contact/:id
* **Controller:** deleteContact
* **Purpose:** Remove a specific contact from DB.

Example:

DELETE /api/contact/65128ad9f3...

**(e) Get Single Contact by ID**

router.get("/:id", getContactById);

* **Method:** GET
* **URL:** /api/contact/:id
* **Controller:** getContactById
* **Purpose:** Fetch details of one specific contact.

Example:

GET /api/contact/65128ad9f3...

**4. module.exports = router;**

* Exports the router so that app.js can use it:
* app.use("/api/contact", contactRoutes);

✅ **Summary**:  
This router defines a **full CRUD API** for contacts:

* POST /api/contact → Create
* GET /api/contact → Read All
* GET /api/contact/:id → Read One
* PUT /api/contact/:id → Update
* DELETE /api/contact/:id → Delete

contactController

**🔑 Import statements**

import moment from "moment-timezone";

import Contact from "../models/Contact.js";

* **moment-timezone** → Used to format timestamps (createdAt, updatedAt) into readable date/time in **India Standard Time (Asia/Kolkata)**.
* **Contact** → Your Mongoose model (from Contact.js), which talks to MongoDB.

**1. Create a Contact (POST /api/contact)**

export const createContact = async (req, res) => {

try {

const { name, email, message } = req.body || {};

if (!name || !email || !message) {

return res.status(400).json({ message: "All fields are required" });

}

const newContact = await Contact.create({ name, email, message });

const response = {

id: newContact.\_id,

name: newContact.name,

email: newContact.email,

message: newContact.message,

createdAt: moment(newContact.createdAt)

.tz("Asia/Kolkata")

.format("D MMMM, YYYY hh:mm A"),

};

res.status(201).json({ message: "Message sent successfully", data: response });

} catch (error) {

res.status(500).json({ message: "Server Error", error: error.message });

}

};

👉 Flow:

1. Extracts name, email, message from req.body.
2. Validates that all fields exist.
3. Creates a new contact in MongoDB with Contact.create().
4. Formats createdAt into human-readable IST (e.g., 25 September, 2025 10:30 AM).
5. Sends back a 201 Created response with the saved contact.

**2. Get All Contacts (GET /api/contact)**

export const getContacts = async (req, res) => {

try {

const contacts = await Contact.find().sort({ createdAt: -1 });

res.status(200).json({

success: true,

count: contacts.length,

data: contacts.map((c) => ({

id: c.\_id,

name: c.name,

email: c.email,

message: c.message,

createdAt: moment(c.createdAt)

.tz("Asia/Kolkata")

.format("D MMMM, YYYY hh:mm A"),

})),

});

} catch (error) {

res.status(500).json({ message: "Server Error", error: error.message });

}

};

👉 Flow:

1. Fetches all contacts (Contact.find()) sorted **newest first**.
2. Returns total count + an array of formatted contacts.

**3. Update Contact (PUT /api/contact/:id)**

export const updateContact = async (req, res) => {

try {

const contactId = req.params.id;

const { name, email, message } = req.body || {};

if (!name && !email && !message) {

return res.status(400).json({ message: "At least one field is required to update" });

}

const contact = await Contact.findById(contactId);

if (!contact) {

return res.status(404).json({ message: "Contact not found" });

}

if (name) contact.name = name;

if (email) contact.email = email;

if (message) contact.message = message;

await contact.save();

const response = {

id: contact.\_id,

name: contact.name,

email: contact.email,

message: contact.message,

createdAt: moment(contact.createdAt).tz("Asia/Kolkata").format("D MMMM, YYYY hh:mm A"),

updatedAt: moment(contact.updatedAt).tz("Asia/Kolkata").format("D MMMM, YYYY hh:mm A"),

};

res.status(200).json({ message: "Contact updated successfully", data: response });

} catch (error) {

res.status(500).json({ message: "Server Error", error: error.message });

}

};

👉 Flow:

1. Gets contact ID from req.params.id.
2. Checks if there’s at least one field to update.
3. Finds contact by ID (findById).
4. Updates only the provided fields.
5. Saves changes and returns formatted response with updatedAt.

**4. Delete Contact (DELETE /api/contact/:id)**

export const deleteContact = async (req, res) => {

try {

const contactId = req.params.id;

const contact = await Contact.findById(contactId);

if (!contact) {

return res.status(404).json({

success: false,

message: "Contact not found",

});

}

await Contact.findByIdAndDelete(contactId);

res.status(200).json({

success: true,

message: "Contact deleted successfully",

});

} catch (error) {

res.status(500).json({

success: false,

message: "Server Error",

error: error.message,

});

}

};

👉 Flow:

1. Reads id from URL.
2. Checks if contact exists.
3. Deletes using findByIdAndDelete.
4. Responds with success message.

**5. Get Single Contact (GET /api/contact/:id)**

export const getContactById = async (req, res) => {

try {

const contactId = req.params.id;

const contact = await Contact.findById(contactId);

if (!contact) {

return res.status(404).json({

success: false,

message: "Contact not found",

});

}

const response = {

id: contact.\_id,

name: contact.name,

email: contact.email,

message: contact.message,

createdAt: moment(contact.createdAt).tz("Asia/Kolkata").format("D MMMM, YYYY hh:mm A"),

updatedAt: moment(contact.updatedAt).tz("Asia/Kolkata").format("D MMMM, YYYY hh:mm A"),

};

res.status(200).json({

success: true,

data: response,

});

} catch (error) {

res.status(500).json({

success: false,

message: "Server Error",

error: error.message,

});

}

};

👉 Flow:

1. Reads contact ID from URL.
2. Finds contact in DB (findById).
3. If found, returns a single formatted contact with createdAt & updatedAt.

**✅ Summary**

* **createContact (POST)** → Add new contact (validates fields).
* **getContacts (GET)** → List all contacts (sorted by time).
* **updateContact (PUT)** → Update specific contact fields.
* **deleteContact (DELETE)** → Remove a contact.
* **getContactById (GET)** → Get details of one contact.

All timestamps are nicely formatted in **Indian timezone** for readability.