MERN Stack

- with project Exercise Tracker App

Table of Contents

---

1. Introduction

Welcome to building web apps with the MERN stack (MongoDB, Express, React, Node.js), using MongoDB Atlas and Google Cloud Platform. In this tutorial, we will create an **Exercise Tracker** app.

Outline:

1. Introduction to the MERN stack
2. Database concepts (MongoDB vs. relational)
3. MongoDB Atlas setup
4. Backend (Node.js + Express + Mongoose)
5. Testing API with Insomnia/Postman
6. Frontend (React + React Router + Axios)
7. Connecting frontend to backend

Teacher: Beau Carnes (freeCodeCamp)

---

2. Technology Stack

- **MongoDB**: Document-based open-source database
- **Express**: Web application framework for Node.js, help to create server, lightweight, fast
- **React**: Front-end JavaScript library for building UIs
- **Node.js**: JavaScript runtime environment that executes JS code outside of the browser (server)
- **Mongoose**: Simple, schema based solution to model application data. ODM (Object Data Modeling) library for MongoDB + Node.js
- **CORS**: Cross-Origin Resource Sharing middleware, helps to AJAX request to skip same origin policy and access resources from remote hosts. interlinked with Express middleware
- **dotenv**: Load environment variables from `.env` file into process.env, makes dev easier

---

3. Database Concepts

Relational (SQL)	MongoDB (NoSQL)
Database	Database
Tables	Collections
Rows	Documents

Relational (SQL)	MongoDB (NoSQL)
Columns	Fields
Foreign keys	References (using ObjectIDs)
JOINs	`$lookup` operator
Data on disk: tables	Data on disk: **BSON** (binary JSON) — supports strings, integers, dates, etc.

> BSON

- looks like json
- store in disc as bson format
- wide variety datatype supports
- allows nesting documents
- improves data integrity instead of splitting into different tables

> Example of a MongoDB document(row) Model and labels the different BSON data types used

```
{
  name: "Beau Carnes",                      // String
  title: "Developer & Teacher",            // String
  address: {                               // Nested Document
    address_1: "123 Main Street",          // String
    city: "Grand Rapids",                  // String
    state: "Michigan",                     // String
    postal_code: "49503"                   // String
  },
  topics: ["MongoDB", "Python", "JavaScript", "Robots"],  // Array of Strings
  employee_number: 1234,                   // Integer
  location: [44.9901, 123.0262]            // Geo-Spatial Coordinates (Array of Numbers)
}
```

> **Subdocuments** and **arrays** allow nesting related data together for faster access.

📚 MongoDB vs Mongoose — Explained Simply

Term	Excel Analogy	Simple Explanation
Database	A workbook (Excel file)	A container holding many collections (like multiple sheets in Excel)
Collection	A single sheet in Excel	A group of similar data (e.g., `users`, `products`)
Document	A single row in Excel	A single data entry (e.g., one user) stored in JSON format

Term	Excel Analogy	Simple Explanation
Field	A cell or column label	A key/value pair inside a document (e.g., `username: "robin"`)
_id	Row number (auto)	Unique ID automatically given to every document
JSON / BSON	Structured data	Format MongoDB uses to store and exchange data (BSON = Binary JSON)

🛠 Mongoose Essentials

Term	Excel Analogy	Simple Explanation
Schema	Column layout & rules	Defines structure of documents (what fields, what type, required or not)
Model	Excel operations tool	A reusable object to create, read, update, delete documents in a collection
Instance / Object	A new row	A new document created from a model
Validator	Data rule checker	Ensures data is correct (e.g., username must be at least 3 characters)
Middleware	Auto-checklist before saving	Code that runs before or after saving documents (like pre-checks)
Population	VLOOKUP	Replace reference IDs with real documents (like showing full user info in an order)

🔄 CRUD Operations (Actions on Data)

Operation	Excel Action	MongoDB Method	What It Does
Create	Add new row	`insertOne`, `Model.create()`	Add a new document
Read	View rows	`find()`, `findOne()`	Fetch documents
Update	Edit a row	`updateOne`, `findByIdAndUpdate()`	Change data in a document
Delete	Remove row	`deleteOne`, `findByIdAndDelete()`	Remove a document

⚙ Example in Code

```
const userSchema = new mongoose.Schema({
  username: { type: String, required: true, minlength: 3 }
});
```

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

const newUser = new User({ username: 'robin' });
newUser.save(); // Adds new document to MongoDB
```

🧠 TL;DR Quick Summary Table

Concept	MongoDB	Mongoose
Database	✅	❌
Collection	✅	❌
Document	✅	✅
Schema	❌	✅
Model	❌	✅
CRUD operations	✅	✅ (simplified)
Validation	Basic	Powerful & easy
Relations	Manual refs	Population feature

---

4. MongoDB Atlas Setup

1. **Sign in** to MongoDB Atlas at `https://cloud.mongodb.com`
2. Click **"New Project"** → name your project → **"Create Project"**
3. Click **"Build a Cluster"** → choose **Google Cloud Platform**, **Free Tier (M0 sandbox)**, select a region → **"Create Cluster"**
4. **Whitelist IP & Create DB User**:
   - In **Network Access**, add your current IP.
   - In **Database Access**, create a user & password.
5. **Get Connection String**:
   - Click **"Connect"** → **"Connect your application"** → copy the connection string.
   - It will look like:

     ```
     mongodb+srv://<username>:
     <password>@cluster0.mongodb.net/myFirstDatabase?
     retryWrites=true&w=majority
     ```

6. **Database & Collections** for this Projectapp:

   - **Collections**:
     - `users`,
     - `exercises`
     - (each exercise references one user)

ObjectIds in MongoDB

Here’s the breakdown of a MongoDB ObjectId as shown:

```
ObjectId("5c2fc4b3 e52f37b7ee a58d00")

  5c2fc4b3   ← 4-byte UNIX timestamp
  e52f37b7ee ← 5-byte random value
  a58d00     ← 3-byte incremental counter
```

- auto generated by Mongodb driver
- is guranteed to be unique across each document in collection
- different part of object id represent different things

---

7. Frontend Setup (React) (Just Initialize)

7.1 Initialize React App

```
# from project root:
npx create-react-app mern-exercise-tracker
cd mern-exercise-tracker
npm install axios react-router-dom bootstrap react-datepicker
```

- `create-react-app` will generate default react project with default dependencies installed
- do `/node_modules` to `node_modules` in `.gitignore`

Remove unused files (logo, serviceWorker) and CSS imports.

5. Backend Setup (Node.js + Express + Mongoose)

5.1 Prerequisites

- **Node.js** installed (`node -v`)
- **npm** available
- **MongoDB Atlas** cluster ready

5.2 Initialize Backend Project

```
# you can either create backend folder inside the frontend directory or
make it seperate adjacent to it
mkdir backend && cd backend
npm init -y #create package.json
npm install express mongoose cors dotenv #backend packages
npm install -g nodemon #makes dev easier, tool to make NodeJS Applications
by automatically restarting the node application when files changes in
directory/detective
```

```
copy .gitignore of frontend to backend
```

nodemon

- makes dev easier,
- tool to make NodeJS Applications
- by automatically restarting the node application when files changes in directory/detective
- SO WHENEVER WE UPDATE OUR SERVER FILE, IT AUTOMATICALLY RESTARTS THE SERVER

> whenever to install globally ,use sudo

5.3 Create `backend/server.js` [BASIC, to Append on the way]

```
// server.js
// body parser not needed in new version of express

// import express framework and core middleware to enable cross-origin
requests
const express = require('express')
const cors = require('cors')

// load env var form .env into process.env
require('dotenv').config();


// create a new express app
const app = express();
// get port no. from environment or default 5000
const port = process.env.PORT || 5000;


// setup middleware
// load cors, now app allows requests from any origin
app.use(cors());

// load express, now app auto parse json payloads in incomin requests
// as our server is gonna send&receive JSONs
app.use(express.json()); // bodyparser is included in express
```

```
// start the server and listen on specific port
app.listen(port, ()=> {console.log(`Server Running SUCCESS at Port :
${port}`)})

// basic server ready :)
```

- run it

```
nodemon server
```

```
bali-king@war-machine:~/BaliGit/fullstack-webdev-essentials/MERN/backend$
nodemon server
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
[dotenv@17.2.1] injecting env (1) from .env -- tip: ⚙  suppress all logs
with { quiet: true }
Server Running SUCCESS at Port : 5000
```

- now we are ready to connect mongodb atlas with server

5.4 Environment Variable

Create `.env` in `/backend`:

```
ATLAS_URI=mongodb+srv://<username>:
<password>@cluster0.mongodb.net/exercise-tracker?
retryWrites=true&w=majority
```

> type in password and username :0 on : never store `.env` file in git repo or getfkd. put them in `.gitignore`

```
.env
.env.local
.env.development.local
.env.test.local
.env.production.local
```

Integrate Mongoose in `backend/server.js` to connect mongodb atlas with server

```
// server.js
// body parser not needed in new version of express

// import express framework and core middleware to enable cross-origin
requests
const express = require('express')
const cors = require('cors')

// ### Integrate Mongoose in `backend/server.js` to connect mongodb atlas
with server
// import mongoose for connecting to mongodb
const mongoose = require('mongoose')

// load env var form .env into process.env
require('dotenv').config();


// create a new express app
const app = express();
// get port no. from environment or default 5000
const port = process.env.PORT || 5000;


// setup middleware
// load cors, now app allows requests from any origin
app.use(cors());

// load express, now app auto parse json payloads in incomin requests
// as our server is gonna send&receive JSONs
app.use(express.json()); // bodyparser is included in express

// after setting up middleware
// load mongodb conn. string from .env
// we will get uri from mongodb atlas dashboard
// uri, where our db is stored
const uri = process.env.ATLAS_URI;

// tell mongoose to connect mongodb using that uri
// As of Mongoose 6.x, many connection options (including useCreateIndex,
useNewUrlParser, useUnifiedTopology, etc.) are now set by default or
deprecated.
mongoose.connect(uri)  .then(() => console.log("MongoDB db-connection est.
SUCCESS"))
  .catch((err) => console.error("MongoDB connection ERROR:", err));



// old way
// // tell mongoose to connect mongodb using that uri
// mongoose.connect(uri, { // passing uri

//     // extra flags, due to internal mongodb update guidelines
//     useNewUrlParser: true, // use the new URL parser instead of the
deprecated one
//     useCreateIndex: true // use createIndex() instead of ensureIndex()
// })

// grab default connection obj
const connection = mongoose.connection;


// old way to detect
// replaced by .then(() => console.log("MongoDB db-connection est.
SUCCESS")) .catch((err) => console.error("MongoDB connection ERROR:",
err));
// // listen,1st time the connection opens, log est success
// connection.once('open',
//     ()=>{
//         console.log("MongoDB db-connection Est. SUCCESS")
//     }
// )

// start the server and listen on specific port
app.listen(port, ()=> {console.log(`Server Running SUCCESS at Port :
${port}`)})

// server+mongoose server ready :)
```

```
bali-king@war-machine:~/BaliGit/fullstack-webdev-essentials/MERN/backend$
nodemon server
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
[dotenv@17.2.1] injecting env (1) from .env -- tip: ⚙️  load multiple .env
files with { path: ['.env.local', '.env'] }
Server Running SUCCESS at Port : 5000
MongoDB db-connection est. SUCCESS
^Cbali-king@war-machine:~/BaliGit/fullstack-webdev-essentials/MERN/backend$
```

5.5 Mongoose Models

- now let's setup db
- let's make schema using mongoose
- let's make `backend/model` folder ,where we have `user.model.js`,`excercise.model.js`

> In Mongoose, a model is a blueprint for a collection in MongoDB.Once you define a model, you can use it to: Create, Read, Update, and Delete documents in that collection (CRUD) If MongoDB is like a database of Excel sheets, then a Mongoose model is like defining the columns and rules for one sheet.

`models/user.model.js`

- 1 field
- multiple validations

```
const mongoose = require('mongoose');// Import the mongoose library to work
with MongoDB
const Schema = mongoose.Schema;// Get the Schema constructor from mongoose
to define data structure
const userSchema = new Schema( // Define a new schema (structure) for a
"User" document
    // single field `username`

    {
        username :   // "username" field must be a string
        {

            // validators
            type: String, // Data type is String
            required: true, // This field is mandatory
            unique:true, // No two users can have the same username
            trim: true,  // Removes extra spaces at the beginning or end
            minlength: 3 // Must be at least 3 characters long
        }

    }
    ,
    {
        timestamps:true  // Automatically adds "createdAt" and "updatedAt"
fields
    }
);

const User = mongoose.model('User',userSchema); //mongoose.model() creates
a model named 'User' using the schema userSchema.
module.exports = User; //This line exports the User model so you can import
and use it in other files (e.g., routes or controllers).
```

`models/exercise.model.js`

- now lets exercise :0
- 4 fields
- less validations
- same as user.model.js

```
const mongoose = require('mongoose')
const Schema =  mongoose.Schema;

const excerciseSchema = new Schema(
    {
        username : {type: String, required: true},
        description: {type: String, required: true},
        duration: {type: Number, required:true},
        date:{type:Date, required:true},

    }
    ,
    {
        timestamps: true,
    }
);

const Exercise = mongoose.model('Exercise',excerciseSchema);
module.exports = Exercise;
```

- now models created

5.6 Routes

- now we have to add API endpoint routes
- so that server could do `CRUD` operations
- mkdir routes && cd routes
- touch exercises.js users.js
- before making it
- we are doing server work

importing and using `routes/users.js` & `routes/exercises.js` in `server.js`

```
// server.js
// body parser not needed in new version of express

// import express framework and core middleware to enable cross-origin
requests
const express = require('express')
const cors = require('cors')



// ### Integrate Mongoose in `backend/server.js` to connect mongodb atlas
with server
// import mongoose for connecting to mongodb
const mongoose = require('mongoose')

// load env var form .env into process.env
```

```
require('dotenv').config();


// create a new express app
const app = express();
// get port no. from environment or default 5000
const port = process.env.PORT || 5000;


// setup middleware
// load cors, now app allows requests from any origin
app.use(cors());

// load express, now app auto parse json payloads in incomin requests
// as our server is gonna send&receive JSONs
app.use(express.json()); // bodyparser is included in express

// after setting up middleware
// load mongodb conn. string from .env
// we will get uri from mongodb atlas dashboard
// uri, where our db is stored
const uri = process.env.ATLAS_URI;

// tell mongoose to connect mongodb using that uri
// As of Mongoose 6.x, many connection options (including useCreateIndex,
useNewUrlParser, useUnifiedTopology, etc.) are now set by default or
deprecated.
mongoose.connect(uri)  .then(() => console.log("MongoDB db-connection est.
SUCCESS"))
  .catch((err) => console.error("MongoDB connection ERROR:", err));




// old way
// // tell mongoose to connect mongodb using that uri
// mongoose.connect(uri, { // passing uri

//     // extra flags, due to internal mongodb update guidelines
//     useNewUrlParser: true, // use the new URL parser instead of the
deprecated one
//     useCreateIndex: true // use createIndex() instead of ensureIndex()
// })

// grab default connection obj
const connection = mongoose.connection;


// old way to detect
// replaced by .then(() => console.log("MongoDB db-connection est.
SUCCESS")) .catch((err) => console.error("MongoDB connection ERROR:",
err));
// // listen,1st time the connection opens, log est success
// connection.once('open',
//     ()=>{
```

```
//         console.log("MongoDB db-connection Est. SUCCESS")
//     }
// )



// #### importing and using `routes/users.js` & `routes/exercises.js` in
`server.js`
// just before app.listen()
// import routes
const exercisesRouter = require('./routes/exercises');
//exercisesRouter is now an Express router containing endpoints like POST,
GET, etc., for exercises.
const userRouter = require('./routes/users'); //usersRouter will handle
routes related to user operations like registration or listing users

app.use('/exercises', exercisesRouter); // Mount exercise routes at
/exercises
// This tells the Express app to use all routes from exercisesRouter, and
prefix them with /exercises.
// For example, if exercisesRouter has a GET / route, it will be available
at GET /exercises/.
app.use('/users',userRouter) // Mount user routes at /users


// start the server and listen on specific port
app.listen(port, ()=> {console.log(`Server Running SUCCESS at Port :
${port}`)})

// server+mongoose server ready :)
```

- don't run it till you makes routes :0

`routes/users.js`

```
const Express = require('express'); // import express

const router = Express.Router();
// NOT // const Router = Express.Router; as Router var is conflict Router
stuff // use router func to create route handlers
let User = require('../models/user.model'); // import model/mongoose schema

// 1st Route/endpoint that handles http get req/
// Route to GET Req. to fetch all users
router.route('/').get( //get request at index
    (req, res) =>
    {
        User.find()   // Find all user records from the database, method of
mongoose, it returns a promise
        .then(users => res.json(users))    // If successful, return the
users in JSON format
        .catch(err=> res.status(400).json('ERROR: '+err));     // If error occurs, send a 400 status with the error message
    }
);

// req ,from Express, This object represents the incoming HTTP request.
// res ,from Express, This object is used to send the response back to the client.

// Define a route for POST request to add a new user
router.route('/add').post(
    (req,res) => {
        const username = req.body.username;   // Extract username from the request body
        // req.body.username: This gets the "username" field from the data sent by the client (like a form or Postman)
        // For example: if you send { "username": "robin" }, this line will store "robin" in the variable `username`
        const newUser = new User({username});   // Create a new User object using the username
        // This creates a new object (document) using the User model
        // The new object looks like: { username: 'robin' }

        newUser.save()   // Save the new user to the database
        // .save(): This is a Mongoose method that saves the new user to the MongoDB database

        .then(()=>res.json('User added!'))     // If successful, respond with a success message
        .catch(err=> res.status(400).json('ERROR: '+err));     // If error occurs, send a 400 status with the error message
    }


);

module.exports = router;
// Export the router so it can be used in other parts of the app
```

> req ,from Express, This object represents the incoming HTTP request. res ,from Express, This object is used to send the response back to the client. An HTTP request is a message sent by a client (like a browser, Postman, or frontend app) to a server, asking it to do something. Great question!

📦 `req` **(Request)**

- Contains information about the HTTP request made by the **client (browser, frontend app, etc)**.
- Includes data like:

  - `req.body` → data sent by the client (POST/PUT)
  - `req.params` → URL parameters (e.g., `/user/:id`)
  - `req.query` → query string (e.g., `?search=apple`)
  - `req.headers` → metadata about the request

👉 Example:

```
const username = req.body.username;
// This gets the 'username' field sent in the body of a POST request
```

- common req methods | Syntax | Purpose | | ------------ | -------------------------------------- | | `req.body` | Data sent in the body (POST, PUT) | | `req.params` | Route parameters (e.g., `/user/:id`) | | `req.query` | Query string (e.g., `/search?term=car`) | | `req.headers` | All the headers from the client | | `req.method` | HTTP method (GET, POST, etc.) | | `req.url` | Requested URL |

📤 `res` (Response)

- Used to **send back a response** to the client.

- Can be a:

  - JSON object → `res.json({ message: "Done" })`
  - Status code → `res.status(400)`
  - Plain text → `res.send("Hello")`

👉 Example:

```
res.json("User added!");
// This sends a JSON response back to the client
```

- common res methods

Syntax	Purpose
`res.send(data)`	Send a plain text or HTML response
`res.json(data)`	Send a JSON response
`res.status(code)`	Set HTTP status code (e.g., 200, 404, 500)
`res.redirect(url)`	Redirect to another URL
`res.end()`	End the response process

> arrow function is just shortcut to write typical function

```
// // Long Way (Traditional Function)
// function(req, res) {
//   // code
// }

// // Short Way (Arrow Function)
// (req, res) => {
//   // code
// }
```

`routes/exercises.js`

- similar theory to users.js
- but slight different code

```
// Import express and create a router object
const Express = require('express');

const router = Express.Router(); //() is imp as This will assign the Router
function itself, not an instance.
// NOT // const Router = Express.Router; as Router var is conflict Router
stuff
// Import the Exercise model
const Exercise = require('../models/excercise.model');

// MyChull :)
const INDEX = router.route('/');
const ADD = router.route('/add');
const ID = router.route('/:id');
const UPDATE_ID = router.route('/update/:id');

// GET all exercises
// Route: GET '/'
// Find all exercises and send as JSON
INDEX.get(function(req,res){
    Exercise.find()
    // Non Arrow function MyChull :)
    .then(function(exercises){
        return res.json(exercises)
    })
    .catch(err=> res.status(400).json('ERROR: '+err));     // If error
occurs, send a 400 status with the error message

});

// POST: Add a new exercise
// Route: POST '/add'
// Extract username, description, duration, date from req.body
// Create a new Exercise object
// Save it to database
// Send success or error response
ADD.post(function(req,res){

    // MyCHull :)
    const REQ = req.body;
    const username = REQ.username,
    description = REQ.description,
    duration = Number(REQ.duration), // Number() Convert duration to a
number

    date = Date.parse(REQ.date); // Date.parse() Convert date string to
Date format
    // cleaner way
    // const {username,description,duration,date} = req.body;

    // Destructure fields from the request body (sent by the client)
    const newExercise = new Exercise (
        {
            username,
            description,
            duration,
            date
        }

    );

    newExercise.save() //save new exercise to mongodb
    .then(function(){res.json("Exercise added !")})
    .catch(function(err){res.status(400).json('Error: '+err)})
    ;

});


// GET: Fetch single exercise by ID
// Route: GET '/:id'
// Find exercise by ID from URL
// Return exercise or send error
ID.get(function(req,res){ //function(res, req) { ... },no swap req,res ✖
ORDER MATTERS, TYPICAL READ OF ARGUEMENTS
    Exercise.findById(req.params.id)// Excercise is mongoose model
representin Exercise colleciton in MongoDB db
    // findById is a Mongoose method It searches for a document by its
unique _id field (the default MongoDB ID for every document).
    .then(function(Exercise){res.json(Exercise)})
    .catch(function(err){res.status(400).json('Error:'+ err)})
    ;

});


// DELETE: Remove exercise by ID
// Route: DELETE '/:id'
// Delete the exercise by ID
// Send success or error response
ID.delete(function(
    req,res
){
    Exercise.findByIdAndDelete(
        req
        .params
        .id
    )// Delete exercise with matching ID
    .then(() => res.json('Exercise deleted !'))  // Respond with success
message
    .catch(err => res.status(400).json('Error: ' + err));  // Handle error


}
);


// POST: Update an existing exercise
// Route: POST '/update/:id'
// Find exercise by ID
// Update its fields with values from req.body
// Save the updated exercise
// Send success or error response
UPDATE_ID.post((req,res)=>{
    Exercise.findById(req.params.id)
    .then(
        function(exercise){
            exercise.username = req.body.username;
            exercise.description = req.body.description;
            exercise.duration = Number(req.body.duration);
            exercise.date = Date.parse(req.body.date);

            exercise.save()
            .then(() => res.json('Exercise Updated !'))  // Respond with
success message
            .catch(err => res.status(400).json('Error: ' + err));  //
Handle error

        }
    )
    .catch(err => res.status(400).json('Error: ' + err));  // Handle error

});



// Export the router so it can be used in server.js
module.exports = router;
```

Params

- parameters
- special rewuest in express request
- stores route parameters from URL Path
- access through `req.params`
- basically access stuff from url :0
- now lets test our APIs

---

6. Testing API (Insomnia / Postman)

- open API Tester
- creat collection
- make request
- shoot it :0
- SUCCESS MEANS `200`

1. **POST** `http://localhost:5000/users/add`

   ```
   { "username": "Momotaro" }
   ```

   output : `"User added!"`

2. **GET** `http://localhost:5000/users`

   output :

   ```
   [
     {
         "_id": "688a6c7c2ea7acb852ea682a",
         "username": "momotaro",
         "createdAt": "2025-07-30T19:03:24.994Z",
         "updatedAt": "2025-07-30T19:03:24.994Z",
         "__v": 0
     },
     {
         "_id": "688a6cf72ea7acb852ea682c",
         "username": "bali",
         "createdAt": "2025-07-30T19:05:27.299Z",
         "updatedAt": "2025-07-30T19:05:27.299Z",
         "__v": 0
     },
     {
         "_id": "688a6cfb2ea7acb852ea682e",
   ```

```
        "username": "bhati",
        "createdAt": "2025-07-30T19:05:31.711Z",
        "updatedAt": "2025-07-30T19:05:31.711Z",
        "__v": 0
    },
    {
        "_id": "688a6d012ea7acb852ea6830",
        "username": "bhaskar",
        "createdAt": "2025-07-30T19:05:37.677Z",
        "updatedAt": "2025-07-30T19:05:37.677Z",
        "__v": 0
    }
]
```

- mongodb auto created _id, and other stuff
- also we save to db !!
- check from mongodb>atlas>exercise-tracker>collections
- refresh if any lag

3. **POST** `http://localhost:5000/exercises/add`

```
 {
   "username":"bali",
   "description":"Jumping Jacks",
   "duration": "9" ,
   "date":"2025-08-30T19:03:24.994Z"
 }
```

output : `"Exercise added !"`

4. **GET** `http://localhost:5000/exercises`

output :

```

[
    {
        "_id": "688a6fee71069ae17aa9e424",
        "username": "bali",
        "description": "Mountain Climbing",
        "duration": 9,
        "date": "2025-08-30T19:03:24.994Z",
        "createdAt": "2025-07-30T19:18:06.596Z",
        "updatedAt": "2025-07-30T19:18:06.596Z",
        "__v": 0
    },
    {
        "_id": "688a6ffb71069ae17aa9e426",
        "username": "bali",
        "description": "Jumping Jacks",
        "duration": 9,
        "date": "2025-08-30T19:03:24.994Z",
        "createdAt": "2025-07-30T19:18:19.287Z",
        "updatedAt": "2025-07-30T19:18:19.287Z",
        "__v": 0
    },
    {
        "_id": "688a707071069ae17aa9e428",
        "username": "bhati",
        "description": "Deadlift",
        "duration": 11,
        "date": "2025-08-30T19:03:24.994Z",
        "createdAt": "2025-07-30T19:20:16.402Z",
        "updatedAt": "2025-07-30T19:20:16.402Z",
        "__v": 0
    }
  ]
```

5. **GET** `http://localhost:5000/exercises/<id>`

   - object id auto created by mongodb
   - accessing `/:id` object id from database, then it will return that info only, directly access stuff from putting id in url
   - eg: `http://localhost:5000/exercises/688a6fee71069ae17aa9e424`
   - output :

```
    {
    "_id": "688a6fee71069ae17aa9e424",
    "username": "bali",
    "description": "Mountain Climbing",
    "duration": 9,
    "date": "2025-08-30T19:03:24.994Z",
    "createdAt": "2025-07-30T19:18:06.596Z",
    "updatedAt": "2025-07-30T19:18:06.596Z",
    "__v": 0
    }
```

6. **POST** `http://localhost:5000/exercises/update/<id>` (with updated fields)

   - eg: `http://localhost:5000/exercises/update/688a6fee71069ae17aa9e424`
   - &

```
          {
     "username": "bhaskar",
     "description": "Mountain Climbing",
     "duration": 9,
     "date": "2025-08-30T19:03:24.994Z"
     }
```

```
```

- Output : `"Exercise Updated !"`
- YOU ALWAYS HAVE TO SEND PROPER ALL FIELDS,NOT PARTIAL OF ANY KIND, NOTHING LESS, NOTHING MORE , JUST EXACT !!!!!!

7. **DELETE** `http://localhost:5000/exercises/<id>`

   - literally same as update , just `POST -> DELETE`
   - eg: `http://localhost:5000/exercises/688a6fee71069ae17aa9e424`, not exercises/delete/:id
   - &

   ```
           {
         "username": "bhati",
         "description": "Deadlift",
         "duration": 11,
         "date": "2025-08-30T19:03:24.994Z"
     }
   ```

   - Output : `"Exercise Deleated !"`
   - YOU ALWAYS HAVE TO SEND PROPER ALL FIELDS,NOT PARTIAL OF ANY KIND, NOTHING LESS, NOTHING MORE , JUST EXACT !!!!!!

- all stuff sync with mongodb database !!
- you can directly do changes in MongoDB Atlas Panel !!
- good job :0

---

7. Frontend Setup (React) (Actual Code Start)

React

- Declarative, efficient, flexible JS Lib. for building UIs.
- It lets you compose complex UIs from small and isolated pieces of code called COMPONENTs.
- we use components to tell react, what we want to see the screen.
- when our data change, react will effiently update and re-render our components.
- components takes in parametet called PROPs(Properties)
- and it return a hierarchy of views to display throught the render method
- RENDER METHOD returns a description of what you want to see on screen

7.2 `public/index.html`

- Change `<title>` to **Exercise Tracker**
- Ensure `<div id="root"></div>` remains

7.3 `src/index.js`

```
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

7.4 `src/App.js`

```
import React from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import { BrowserRouter as Router, Route } from 'react-router-dom';
import Navbar from './components/Navbar.component';
import ExerciseList from './components/ExerciseList.component';
import CreateExercise from './components/CreateExercise.component';
import EditExercise from './components/EditExercise.component';
import CreateUser from './components/CreateUser.component';

function App() {
  return (
    <Router>
      <div className="container">
        <Navbar />
        <br />
        <Route path="/" exact component={ExerciseList} />
        <Route path="/edit/:id" component={EditExercise} />
        <Route path="/create" component={CreateExercise} />
        <Route path="/user" component={CreateUser} />
      </div>
    </Router>
  );
}

export default App;
```

7.5 Components

7.5.1 `src/components/Navbar.component.js`

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';

export default class Navbar extends Component {
  render() {
    return (
      <nav className="navbar navbar-dark bg-dark navbar-expand-lg">
        <Link to="/" className="navbar-brand">Exercise Tracker</Link>
        <div className="collpase navbar-collapse">
```

```
          <ul className="navbar-nav mr-auto">
            <li className="navbar-item">
              <Link to="/" className="nav-link">Exercises</Link>
            </li>
            <li className="navbar-item">
              <Link to="/create" className="nav-link">Create
Exercise</Link>
            </li>
            <li className="navbar-item">
              <Link to="/user" className="nav-link">Create User</Link>
            </li>
          </ul>
        </div>
      </nav>
    );
  }
}
```

7.5.2 `src/components/CreateUser.component.js`

```
import React, { Component } from 'react';
import axios from 'axios';

export default class CreateUser extends Component {
  constructor(props) {
    super(props);
    this.state = { username: '' };

    this.onChangeUsername = this.onChangeUsername.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }

  onChangeUsername(e) {
    this.setState({ username: e.target.value });
  }

  onSubmit(e) {
    e.preventDefault();
    const user = { username: this.state.username };
    console.log(user);
    axios.post('http://localhost:5000/users/add', user)
      .then(res => console.log(res.data));

    this.setState({ username: '' });
  }

  render() {
    return (
      <div>
        <h3>Create New User</h3>
        <form onSubmit={this.onSubmit}>
```

```
          <div className="form-group">
            <label>Username:</label>
            <input type="text"
              required
              className="form-control"
              value={this.state.username}
              onChange={this.onChangeUsername}
            />
          </div>
          <div className="form-group">
            <input type="submit" value="Create User" className="btn btn-
primary" />
          </div>
        </form>
      </div>
    );
  }
}
```

7.5.3 `src/components/CreateExercise.component.js`

```
import React, { Component } from 'react';
import DatePicker from 'react-datepicker';
import 'react-datepicker/dist/react-datepicker.css';
import axios from 'axios';

export default class CreateExercise extends Component {
  constructor(props) {
    super(props);
    this.state = {
      username: '', description: '', duration: 0, date: new Date(), users:
[]
    };
    // bind methods
    this.onChangeUsername = this.onChangeUsername.bind(this);
    this.onChangeDescription = this.onChangeDescription.bind(this);
    this.onChangeDuration = this.onChangeDuration.bind(this);
    this.onChangeDate = this.onChangeDate.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }

  componentDidMount() {
    axios.get('http://localhost:5000/users/')
      .then(res => { if (res.data.length > 0) {
        this.setState({
          users: res.data.map(user => user.username),
          username: res.data[0].username
        });
      }})
      .catch(err => console.log(err));
  }
  onChangeUsername(e)    { this.setState({ username: e.target.value }); }
  onChangeDescription(e) { this.setState({ description: e.target.value });
}
  onChangeDuration(e)    { this.setState({ duration: e.target.value }); }
  onChangeDate(date)     { this.setState({ date }); }

  onSubmit(e) {
    e.preventDefault();
    const exercise = {
      username: this.state.username,
      description: this.state.description,
      duration: this.state.duration,
      date: this.state.date
    };
    console.log(exercise);

    axios.post('http://localhost:5000/exercises/add', exercise)
      .then(res => console.log(res.data));

    window.location = '/';
  }

  render() {
    return (
      <div>
        <h3>Create New Exercise Log</h3>
        <form onSubmit={this.onSubmit}>
          <div className="form-group">
            <label>Username: </label>
            <select required className="form-control"
              value={this.state.username}
              onChange={this.onChangeUsername}>
              {this.state.users.map(user => (
                <option key={user} value={user}>{user}</option>
              ))}
            </select>
          </div>
          <div className="form-group">
            <label>Description: </label>
            <input  type="text" required className="form-control"
              value={this.state.description}
              onChange={this.onChangeDescription}
            />
          </div>
          <div className="form-group">
            <label>Duration (in minutes): </label>
            <input type="number" className="form-control"
              value={this.state.duration}
              onChange={this.onChangeDuration}
            />
          </div>
          <div className="form-group">
            <label>Date: </label>
```

```
              <div>
                <DatePicker
                  selected={this.state.date}
                  onChange={this.onChangeDate}
                />
              </div>
            </div>
            <div className="form-group">
              <input type="submit" value="Create Exercise Log" className="btn
btn-primary" />
            </div>
          </form>
        </div>
      );
    }
  }
```

7.5.4 `src/components/ExerciseList.component.js`

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';
import axios from 'axios';

// Functional component for a single exercise row
const Exercise = props => (
  <tr>
    <td>{props.exercise.username}</td>
    <td>{props.exercise.description}</td>
    <td>{props.exercise.duration}</td>
    <td>{props.exercise.date.substring(0,10)}</td>
    <td>
      <Link to={`/edit/${props.exercise._id}`}>edit</Link> |
      <a href="#!" onClick={() => {
props.deleteExercise(props.exercise._id) }}>delete</a>
    </td>
  </tr>
);

export default class ExerciseList extends Component {
  constructor(props) {
    super(props);
    this.deleteExercise = this.deleteExercise.bind(this);
    this.state = { exercises: [] };
  }

  componentDidMount() {
    axios.get('http://localhost:5000/exercises/')
      .then(res => this.setState({ exercises: res.data }))
      .catch(err => console.log(err));
  }
  deleteExercise(id) {
    axios.delete(`http://localhost:5000/exercises/${id}`)
      .then(res => console.log(res.data));
    this.setState({ exercises: this.state.exercises.filter(el => el._id !==
id) });
  }

  exerciseList() {
    return this.state.exercises.map(currentexercise => (
      <Exercise
        exercise={currentexercise}
        deleteExercise={this.deleteExercise}
        key={currentexercise._id}
      />
    ));
  }

  render() {
    return (
      <div>
        <h3>Logged Exercises</h3>
        <table className="table">
          <thead className="thead-light">
            <tr>
              <th>Username</th>
              <th>Description</th>
              <th>Duration</th>
              <th>Date</th>
              <th>Actions</th>
            </tr>
          </thead>
          <tbody>
            { this.exerciseList() }
          </tbody>
        </table>
      </div>
    );
  }
}
```

7.5.5 `src/components/EditExercise.component.js`

```
import React, { Component } from 'react';
import DatePicker from 'react-datepicker';
import 'react-datepicker/dist/react-datepicker.css';
import axios from 'axios';

export default class EditExercise extends Component {
  constructor(props) {
    super(props);
    this.state = { username: '', description: '', duration: 0, date: new
Date(), users: [] };

    // bind methods
    this.onChangeUsername = this.onChangeUsername.bind(this);
    this.onChangeDescription = this.onChangeDescription.bind(this);
    this.onChangeDuration = this.onChangeDuration.bind(this);
    this.onChangeDate = this.onChangeDate.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }

  componentDidMount() {
    // fetch exercise

axios.get(`http://localhost:5000/exercises/${this.props.match.params.id}`)
      .then(res => this.setState({
        username: res.data.username,
        description: res.data.description,
        duration: res.data.duration,
        date: new Date(res.data.date)
      }))
      .catch(err => console.log(err));

    // fetch users
    axios.get('http://localhost:5000/users/')
      .then(res => { if(res.data.length > 0) {
        this.setState({ users: res.data.map(user => user.username) });
      }})
      .catch(err => console.log(err));
  }

  onChangeUsername(e)    { this.setState({ username: e.target.value }); }
  onChangeDescription(e) { this.setState({ description: e.target.value });
}
  onChangeDuration(e)    { this.setState({ duration: e.target.value }); }
  onChangeDate(date)     { this.setState({ date }); }

  onSubmit(e) {
    e.preventDefault();
    const exercise = {
      username: this.state.username,
      description: this.state.description,
      duration: this.state.duration,
      date: this.state.date
    };


axios.post(`http://localhost:5000/exercises/update/${this.props.match.param
s.id}`, exercise)
      .then(res => console.log(res.data));

    window.location = '/';
  }

  render() {
    return (
      <div>
        <h3>Edit Exercise Log</h3>
        <form onSubmit={this.onSubmit}>
          {/* Similar form fields as CreateExercise */}
        </form>
      </div>
    );
  }
}
```

---

8. Running the App

1. **Start backend**:

   ```
   cd backend
   nodemon server.js
   ```

2. **Start frontend**:

   ```
   cd mern-exercise-tracker
   npm start
   ```

3. Visit `http://localhost:3000` to interact with your Exercise Tracker!

---

9. Conclusion

You have now built a full-stack **Exercise Tracker** application using the MERN stack:

- **Backend**: Node.js, Express, MongoDB Atlas, Mongoose
- **Frontend**: React, Axios, React Router, Bootstrap

Feel free to extend this app by:

- Adding authentication (JWT)
- Deploying to Heroku / Netlify
- Enhancing UI/UX

Happy coding! 🚀