modernized Express + Mongoose CRUD app code with detailed **explanation**

📘 `server.js` – Full Code with Explanation

```
const express = require('express');
// ⬅ Import the Express.js framework to build the web server

const mongoose = require('mongoose');
// ⬅ Import Mongoose to interact with MongoDB using models

const bodyParser = require('body-parser');
// ⬅ Middleware to parse incoming JSON and URL-encoded data

const Book = require('./Book.model');
// ⬅ Import the Book model (Schema defined in separate file)

const app = express();
// ⬅ Create an Express application object

const port = 8080;
// ⬅ Define the port where the app will run (http://localhost:8080)

const dbURI = 'mongodb://localhost:27017/rajdb33';
// ⬅ Connection string for local MongoDB (specifies port 27017 and DB name
`rajdb33`)

// MIDDLEWARE
app.use(bodyParser.json());
// ⬅ Automatically parse incoming requests with JSON payload

app.use(bodyParser.urlencoded({ extended: true }));
// ⬅ Allows parsing of URL-encoded form data (like HTML form submission)
```

🛠️ Connect to MongoDB

```
mongoose.connect(dbURI)
  .then(() => console.log('MongoDB connected...'))
  // ⬅ If connection is successful, log success
  .catch(err => console.error('MongoDB connection error:', err));
  // ⬅ Handle connection errors
```

> ✅ **Note:** No need to pass `useNewUrlParser` or `useUnifiedTopology` in Mongoose 6+ — defaults are already set.

🔹 Root Route

```
app.get('/', (req, res) => {
  res.send('happy to be here');
  // ⬅ Simple route to test if server is running
});
```

📚 GET All Books

```
app.get('/book', async (req, res) => {
  try {
    const books = await Book.find({});
    // ⬅ Fetch all documents from the `books` collection

    res.setHeader('Cache-Control', 'no-cache, no-store');
    // ⬅ Prevent caching for up-to-date results

    res.json(books);
    // ⬅ Send the result as JSON response
  } catch (err) {
    res.status(500).send(err.message);
    // ⬅ Handle and return server errors
  }
});
```

📘 GET Book by ID

```
app.get('/book/:id', async (req, res) => {
  try {
    const book = await Book.findById(req.params.id);
    // ⬅ Find a book document by MongoDB `_id`

    res.json(book);
    // ⬅ Return book as JSON
  } catch (err) {
    res.status(500).send(err.message);
  }
});
```

📝 POST – Add New Book

```
app.post('/book', async (req, res) => {
  try {
    const newBook = new Book(req.body);
    // ⬅ Create a new Book object using request body data

    const savedBook = await newBook.save();
    // ⬅ Save it to the database

    res.json(savedBook);
    // ⬅ Send saved book in response
  } catch (err) {
    res.status(500).send(err.message);
  }
});
```

> 💡 **Note:** `req.body` must contain `title`, `author`, and `category` for valid insertion.

🔄 PUT – Update Book by ID

```
app.put('/book/:id', async (req, res) => {
  try {
    const updatedBook = await Book.findByIdAndUpdate(
      req.params.id,
      { $set: { title: req.body.title } },
      { new: true, upsert: true }
    );
    // ⬅ Update title; return new version; create if not found (upsert)

    res.json(updatedBook);
  } catch (err) {
    res.status(500).send(err.message);
  }
});
```

- `upsert: true` — If no book is found, it creates a new one
- `new: true` — Return the updated document instead of the old one

❌ DELETE – Remove Book by ID

```
app.delete('/book/:id', async (req, res) => {
  try {
    const deletedBook = await Book.findByIdAndDelete(req.params.id);
    // ⬅ Delete the book with given ID

    res.json(deletedBook);
    // ⬅ Return the deleted book data
  } catch (err) {
    res.status(500).send(err.message);
  }
});
```

🚀 Start the Server

```
app.listen(port, () => {
  console.log(`App listening on port ${port}`);
  // ⬅ Starts server and listens on port 8080
});
```

🧠 Special Notes

Concept	Notes
`mongoose.connect`	Should always be called before any CRUD operations
`req.params.id`	Dynamic route parameter (e.g. `/book/123`)
`req.body`	Parsed automatically by `body-parser` middleware
`await`	Waits for async DB calls like `.find()` or `.save()`
`.find()`	Returns all matching documents (array)
`.findById()`	Returns single document by `_id`

Concept	Notes
`.findByIdAndUpdate()`	Atomically updates and returns modified document
`.findByIdAndDelete()`	Deletes and returns the document
Model name	Defined in `Book.model.js` – used by `mongoose.model()`

📦 `Book.model.js` – Mongoose Schema (Required)

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

const bookSchema = new mongoose.Schema({
  title: String,
  author: String,
  category: String
});

module.exports = mongoose.model('Book', bookSchema);
```

> ⚠ **Important:** The name `Book` becomes the MongoDB **collection** `books` (auto-pluralized by Mongoose).

🔬 Step-by-Step Execution of Your Code

```
app.get('/book/:id', async (req, res) => {
  try {
    const book = await Book.findById(req.params.id); // ⛔ pause here
    res.json(book);                                  // ✅ resume here after DB
responds
  } catch (err) {
    res.status(500).send(err.message);
  }
});
```

Internally, Node.js does:

1. 🛬 **Request arrives** at `/book/:id`.
2. 📝 Enters your `async` function.
3. 🧠 Hits the `await Book.findById(...)`:
   - The Promise is **still pending**.
   - Node **pauses this function**, saving its state.
   - The remaining code (`res.json(...)`) is moved to the **microtask queue**.
   - Node continues serving **other requests** — it's *non-blocking*.
4. 📬 Once the DB responds:
   - The `Promise` resolves.
   - The paused function is **resumed**.
   - The event loop **picks the next task from the microtask queue** — your remaining code.
   - `res.json(...)` runs and sends the result.

📌 Special Notes

Concept	Behavior
`await` keyword	Suspends execution of that async function until Promise resolves
After `await`	Code is deferred to the **microtask queue**
Event loop	Checks the microtask queue **after every I/O or macro task**, so it can resume async functions quickly
Is it blocking?	❌ No. Only that specific `async` function is paused. Node is free to handle other requests

---

🧠 Analogy (Teacher Edition)

Imagine a teacher (Node.js) is helping a student fetch a book from the library (MongoDB via `await Book.findById`). While waiting:

- 👩‍🏫 The teacher marks the place in the lesson (state saved).
- 📚 Goes to help other students (other incoming HTTP requests).
- 🕓 Once the book comes, the teacher **resumes the lesson exactly where paused**.

📝 Thread Model in Node.js

Component	Runs On	Description
🌀 **Event Loop**	✅ Main thread	Handles callbacks, timers, I/O completions
🧱 **Your JavaScript code**	✅ Main thread	Runs top-level and callback code
📝 **Worker Threads / libuv**	✅ Background threads	Handles heavy I/O: file access, DNS, crypto, etc., off the main thread

So What Does “Non-blocking” Mean?

Even though Node.js runs on a **single main thread**, it offloads **I/O tasks** to background threads provided by the **libuv thread pool** (C++ library underneath Node.js).

🔁 **Example Flow:**

```
js
CopyEdit
fs.readFile('file.txt', (err, data) => {
  console.log("File read done!");
});
console.log("Next line");
```

✅ Explanation:

1. `fs.readFile` is handed off to the background thread.
2. Main thread (event loop) moves on — doesn't wait.
3. When file read finishes, callback is pushed to event loop queue.
4. Event loop executes callback when main thread is free.

🧠 Special Note:

- **CPU-intensive tasks (like loops or encryption)** WILL block the event loop because they're executed on the main thread.
- That’s why such tasks should be done in a **Worker Thread**, `child_process`, or moved to a microservice.

📌 TL;DR

Question	Answer
Is event loop on main thread?	✅ Yes
Does main thread do everything?	❌ No — I/O and timers are delegated
Is Node truly single-threaded?	Mostly — but uses libuv's background threads

🔁 Recap: How Node.js Handles an HTTP Request

Suppose a user makes an HTTP request to your Node.js API (say: `/book/:id`), and that route handler includes an **asynchronous operation** like reading from the database or file.

📝 Behind the Scenes — Step-by-Step:

1. 🔗 HTTP request received

- The **event loop (main thread)** receives the request.
- It enters the corresponding route handler:

```
js
CopyEdit
app.get('/book/:id', async (req, res) => { ... });
```

2. 🧠 Async operation initiated

- You run:

```
js
CopyEdit
const book = await Book.findById(req.params.id);
```

- Node uses **libuv + a thread from its internal thread pool** to execute the DB I/O operation.
- The main thread **does NOT block** — it **saves** the async context (i.e., “when done, resume here”).

3. 💤 Event loop continues

- The request handler is **paused** at the `await` point.
- Event loop is now **free to handle other events** (like another HTTP request or timer).

4. ✅ Result ready in background thread

- Once the DB thread finishes fetching data, it:
  - **Sends a message back to the main thread's event loop** with the result.
  - Event loop queues a **callback** (i.e., "resume the paused `await`").

5. 🔄 Event loop picks up that callback

- When it reaches the resume point in the call stack, it continues execution:

```
js
CopyEdit
res.json(book); // sends HTTP response
```

- This uses the **same `res` object** that was created when the request arrived.

✅ **This is how the right HTTP client gets the correct response!**

🔐 How response (`res`) is preserved

- Node.js uses **closures and async context tracking** to **remember** which `res` object belongs to which request.
- Each `res` is scoped to its own request handler.
- Even when the function "pauses" (via `await`), Node remembers where to continue when the data comes back.

💡 Analogy

Imagine the event loop as a waiter in a restaurant:

1. A customer gives an order (HTTP request).
2. The waiter sends it to the kitchen (background thread).
3. The waiter serves other tables (event loop moves on).
4. When the kitchen is done, it rings a bell (callback).
5. The waiter picks up the dish and gives it back to **the correct customer** using a ticket (closure/context).