

Practical: Blog Post App : PART- 01

Backend Setup Guide



Objective:

Set up a backend server with Express and MongoDB, and implement basic blog CRUD APIs (without authentication for now).

Step 1: Initialize Project

```
npm init -y
npm install express mongoose dotenv cors
npm install nodemon --save-dev
```

This step sets up the base of your Node.js backend application. It involves creating a project configuration file, installing necessary packages, and setting up scripts for development and production.

1. Initialize Project Configuration

Run `npm init -y` to create a `package.json` file. This file stores important metadata about the project such as its name, version, and dependencies. The `-y` flag automatically fills in default values so you don't have to input them manually during setup.

2. Install Core Dependencies

Install the following packages using `npm install express mongoose dotenv cors`:

- **express**: A minimal and flexible Node.js web application framework to build APIs and handle routes.
- **mongoose**: An Object Data Modeling (ODM) library for MongoDB and Node.js that provides schema-based solutions to model your data.
- **dotenv**: Loads environment variables from a `.env` file into `process.env`, keeping sensitive configuration out of your source code.
- **cors**: Enables Cross-Origin Resource Sharing, allowing your backend to respond to requests from different origins (e.g., frontend hosted on another port).

These packages are added as **dependencies** in your `package.json`, meaning they are required to run your application.

3. Install Development Dependency

Install `nodemon` using `npm install nodemon --save-dev`. This tool watches your code for changes and automatically restarts the server, improving development efficiency. The `--save-dev` flag adds it to **devDependencies**, indicating it's only needed during development.

Add to `package.json`:

```
"scripts": {  
  "start": "node server.js",  
  "dev": "nodemon server.js"  
}
```

- `start`: Runs your server with Node.js (used for production).
- `dev`: Runs your server using `nodemon` (used during development for auto-reload).

Key Takeaways

- Use `npm init -y` to quickly set up `package.json`.
- Install essential packages like `express`, `mongoose`, `dotenv`, and `cors` to build robust APIs.
- Use `nodemon` as a development tool to auto-reload the server on changes.
- Use scripts like `npm start` and `npm run dev` to simplify your workflow.
- Keep environment variables and sensitive data outside of your codebase using `.env` and `dotenv`.

Step 2: Folder Structure

Step 3: MongoDB Connection File (`config/db.js`)

```
const mongoose = require("mongoose");  
  
const connectDB = async () => {  
  try {  
    const mongoURI = process.env.NODE_ENV === "production" ? process.env.MONGO_URI_PROD :  
      process.env.MONGO_URI_DEV;  
    await mongoose.connect(mongoURI);  
    console.log("MongoDB connected successfully");  
  } catch (error) {  
    console.error("MongoDB connection failed:", error.message);  
    process.exit(1);  
  }  
};  
  
module.exports = connectDB;
```

MongoDB Connection Setup Explanation

This note explains the logic and purpose behind connecting a Node.js application to MongoDB using Mongoose with environment-based configuration.

1. Importing Mongoose

The first line imports the Mongoose library, which allows structured interaction with MongoDB databases using schemas and models.

2. Defining the connectDB Function

A function called `connectDB` is defined as an `async` function. This allows the use of `await` for handling asynchronous database connection operations more cleanly.

3. Choosing the Correct MongoDB URI

The connection URI is chosen based on the environment:

- If the `NODE_ENV` is set to `"production"`, it uses the `MONGO_URI_PROD` environment variable.
- Otherwise, it defaults to `MONGO_URI_DEV`.

This makes the app flexible for both development and production environments.

4. Connecting to MongoDB

Using `mongoose.connect(mongoURI)`, the app attempts to connect to the selected MongoDB URI.

- If the connection is successful, a success message is logged.
- If the connection fails, an error message is displayed and the process exits with code `1` (failure).

5. Exporting the Function

The `connectDB` function is exported using `module.exports`, so it can be imported and used in other files (like `server.js`).

Key Concepts to Highlight for Students

- Use of environment variables to separate dev and prod configs.
- `async/await` makes asynchronous code easier to read and manage.
- Always handle errors gracefully in DB connection logic.
- Exiting the process on DB failure prevents the app from running in a broken state.
- Use of `module.exports` helps with code modularity and reuse.

This setup ensures a clean, scalable, and environment-aware connection to MongoDB.

Step 4: Setup Express Server (**server.js**)

```
const express = require("express");
const dotenv = require("dotenv");
const cors = require("cors");
const connectDB = require("../config/db");
const blogRoutes = require("../routes/blogRoutes");

dotenv.config();
const app = express();

// Middleware
app.use(cors());
app.use(express.json());

// Routes
app.use("/api/blogs", blogRoutes);

// Connect DB and Start Server
connectDB().then(() => {
  const PORT = process.env.PORT || 5000;
  app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
});
```

This file initializes and starts the Express server, connects to MongoDB, and applies essential middleware and routes.

1. Load Required Packages

- **express**: Sets up the web server.
- **dotenv**: Loads environment variables from **.env** file.
- **cors**: Allows cross-origin requests.
- **connectDB**: Custom function to connect to MongoDB.
- **blogRoutes**: Contains route handlers for blog-related APIs.

2. Environment Configuration

- **dotenv.config()** enables usage of variables like **PORT** and **MONGO_URI** from **.env** file.

3. Initialize Express App

- **const app = express();** sets up the core server instance.

4. Middleware Setup

- `app.use(cors())`: Enables CORS.
- `app.use(express.json())`: Parses incoming JSON request bodies.

5. Routes Mounting

- All blog-related routes are mounted at `/api/blogs`.

6. Connect to MongoDB and Start Server

- `connectDB()` connects to MongoDB before starting the server.
- Server listens on the port defined in `.env` or defaults to 5000.

Key Concept:

Server won't start until MongoDB connection is successful, ensuring database readiness before accepting requests.

Step 5: Environment File (`.env`)

```
MONGO_URI_DEV=mongodb://localhost:27017/blogApp
MONGO_URI_PROD=your_production_connection_string_here
PORT=5000
NODE_ENV=development
```

Environment File (`.env`) Notes

Step 6: Blog Model (`models/Blog.js`)

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

const blogSchema = new mongoose.Schema({
  title: String,
  content: String,
  image: String,
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model("Blog", blogSchema);
```

This file defines the data structure for blog posts using Mongoose Schema, which maps to a MongoDB collection.

1. Import mongoose:

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

Used to define the schema and create the model.

2. Define blogSchema:

- **title**: Title of the blog post.
- **content**: Main body of the blog.
- **image**: URL or path to the blog image.
- **createdAt**: Auto-generated timestamp when the blog is created.

3. Export the model:

```
module.exports = mongoose.model("Blog", blogSchema);
```

Creates a model named **Blog** based on **blogSchema**. This model will interact with the **blogs** collection in MongoDB.

Step 7: Blog Controller (**controllers/blogController.js**)

```
const Blog = require("../models/Blog");

exports.getAllBlogs = async (req, res) => {
  const blogs = await Blog.find().sort({ createdAt: -1 });
  res.json(blogs);
};

exports.getBlogById = async (req, res) => {
  const blog = await Blog.findById(req.params.id);
  if (!blog) return res.status(404).json({ msg: "Blog not found" });
  res.json(blog);
};

exports.createBlog = async (req, res) => {
  const { title, content, image } = req.body;
  const newBlog = await Blog.create({ title, content, image });
  res.status(201).json(newBlog);
};
```

This file defines the logic for handling blog-related API requests. It acts as the bridge between the route definitions and the database model.

1. Import the Blog Model

The **Blog** model is imported to interact with the MongoDB collection for blogs.

2. getAllBlogs

- Fetches all blog entries from the database.
- Sorts them in descending order by creation time (`createdAt: -1`) to show newest first.
- Sends the list as a JSON response.

3. `getBlogById`

- Extracts the blog ID from the request parameters (`req.params.id`).
- Uses `Blog.findById()` to fetch the specific blog.
- Returns a 404 response if the blog is not found.
- Otherwise, returns the blog as JSON.

4. `createBlog`

- Extracts `title`, `content`, and `image` from the request body.
- Uses `Blog.create()` to insert a new blog into the database.
- Returns the newly created blog with a 201 (Created) status.

Step 8: Blog Routes (`routes/blogRoutes.js`)

```
const express = require("express");
const { getAllBlogs, getBlogById, createBlog } = require("../controllers/blogController");

const router = express.Router();

router.get("/", getAllBlogs);
router.get("/:id", getBlogById);
router.post("/", createBlog);

module.exports = router;
```

This file defines the API endpoints (routes) related to blog operations. It uses Express Router to organize route handlers separately.

Key Components:

- **`express.Router()`**: Creates a new router instance to define routes independently of the main application.

- **Controller Functions:** It imports `getAllBlogs`, `getBlogById`, and `createBlog` from the `controllers/blogController.js` file to handle the logic for each route.

Defined Routes:

- `GET /api/blogs/` → Calls `getAllBlogs` to fetch all blog posts.
- `GET /api/blogs/:id` → Calls `getBlogById` to fetch a specific blog post by its ID.
- `POST /api/blogs/` → Calls `createBlog` to create a new blog post with data from the request body.

Export

- The configured router is exported and later used in `server.js` via `app.use("/api/blogs", blogRoutes);`

This modular approach helps keep routes clean and maintainable.
