Notes-Production-setup

Backend Architecture: A Structured Approach

We're building a backend application using a simplified version of the Model-View-Controller (MVC) architecture. This helps us organize our code and keeps things maintainable. It's like having different departments in a company, each with a specific job. Here's how it breaks down for our Node.js project:

Core Idea:

- **Separation of Concerns:** The main principle is to separate different aspects of the application:
  - **Server Setup & Entry Point:** `server.js` - Handles server startup and initial setup.
  - **Application Logic and Configuration:** `src/app.js` - Manages Express server configuration, middleware.
  - **Routing:** `src/routes/index.routes.js` - Defines the available API endpoints.
  - **Controllers:** `src/controllers/index.controller.js` - Contains functions that handle specific route logic and business logic.

File-by-File Breakdown:

1. `server.js` **(The Entry Point):**
   - **Role:** This is the starting point of your backend application. It's like the front door of your house.
   - **Responsibilities:**
     - **Server Startup:** Starts the Node.js server, making your application available to handle requests.
     - **Port Setup:** Configures which port your server listens on (e.g., port 3000).
     - **Database Connection (Later):** Will eventually handle the connection to your database when you're ready to add persistence.
     - **Import and use express app:** Import the express app configured in `src/app.js` and use it to handle the incoming requests

- **Analogy:** The building's foundation and main power switch.
- **Key Code:**

```
const express = require('./src/app.js'); // Import ex
press app from src/app.js
const port = 3000;

const app = express(); // Use the express app

app.listen(port, () => {
    console.log(`Server started on port ${port}`);
});
```

2. `src/app.js` **(Application Configuration):**
   - **Role:** This file sets up and configures your Express.js application, kind of like setting up the core components of the system.
   - **Responsibilities:**
     - **Express Setup:** Creates an instance of the Express server.
     - **Middleware:** Adds and configures middleware like:
       - `express.json()`: Parses incoming request bodies as JSON.
     - **Routes setup:** Adds and configures the routes.
     - **Error Handling and 404 Route**: It handles default error messages in case of non existent routes.
   - **Analogy:** The control panel of a machine, setting its core functions.
   - **Key Code:**

```
const express = require('express');
const indexRoutes = require('./routes/index.routes')
// Import the routes


const app = express(); // Create a new express applic
ation
```

```
//Middleware
app.use(express.json()); // For parsing JSON request
bodies
app.use("/",indexRoutes); // Use the routes to handle
request
app.use((req, res)=>{ // Middleware for handling non
existent routes
     res.status(404).json({message: "Not Found"});
});


module.exports = app; // Export the express app
```

3. `src/routes/index.routes.js` **(API Endpoints):**
   - **Role:** This file defines the API endpoints or routes of your application. Think of it as the road map of your app.
   - **Responsibilities:**
     - **Route Definitions:** Specifies which URLs (e.g., `/hello`, `/todos`) will trigger specific actions in your application, and which HTTP method the route is for.
     - **Controller Linking:** Connects each defined route to its corresponding controller function (the function that will handle the logic of this route).
   - **Analogy:** A city map showing which roads lead to which buildings.
   - **Key Code:**

```
const express = require('express');
const controller = require('../controllers/index.cont
roller'); // Import the controller functions

const router = express.Router(); // Create a new rout
er

router.get('/hello', controller.hello); // Define the
/hello route
router.get('/todos', controller.getTodos); // Define
```

```
the /todos get route
router.post('/todos', controller.createTodo); // Defi
ne the /todos post route

module.exports = router; // export the router
```

4. `src/controllers/index.controller.js` **(Business Logic):**
   - **Role:** This file contains the functions (controllers) that handle the logic for each route. This is where most of your application's logic goes.
   - **Responsibilities:**
     - **Request Handling:** Receives the request from the routes and processes the incoming data.
     - **Business Logic:** Executes the application's business rules and functionalities (e.g., fetching data, validating input).
     - **Response Generation:** Sends back an appropriate response to the client (e.g., JSON data, error messages).
   - **Analogy:** The specific workers inside the building, carrying out their tasks.
   - **Key Code Example:**

```
//In src/controllers/index.controller.js
const todos = []; // Store todos (temporary)

 const hello =  (req, res) => {
    res.status(200).json({ message: "Hello World" });
};

const getTodos = (req, res) => {
   res.status(200).json(todos);
};

const createTodo = (req, res) => {
   const {text} = req.body; // grab the text paramete
r from the request body
   if(!text){
```

```
        return res.status(400).json({message: "Please
provide a text for the todo"}) // return a bad reques
t if the text parameter is missing
    }
    const newTodo = { text, completed: false };
    todos.push(newTodo); // Add todo to array
    res.status(201).json(newTodo); // Send new todo b
ack as response
};


module.exports = {hello, getTodos, createTodo}; // Ex
port controller functions
```

Key Takeaways:

- **Organization:** This structure keeps your code organized and easy to manage.
- **Modularity:** Each file has a specific job, making it easier to change or extend functionality.
- **Maintainability:** If you need to change something, you know exactly where to look.
- **Scalability:** This setup is scalable and can handle more complex applications.

Remember:

- This is a simplified MVC-like structure. In larger projects, you might have a separate 'models' folder for database interaction.
- This structure promotes best practices and is a great foundation for building more robust backend applications.

Does this breakdown of the architecture and file roles make sense? Do you have any other questions about it?