### **Standard Operating Procedure (SOP) for Node.js REST API Folder Structure with TypeScript**

This SOP outlines the best practices for structuring a **Node.js REST API** using **TypeScript** to maintain scalability, maintainability, and efficiency.

## **📌 Folder Structure Overview**

project-root/

│── src/

│ ├── config/ # Configuration files (e.g., environment variables, database connections)

│ ├── controllers/ # Route controllers for handling requests and responses

│ ├── middlewares/ # Custom middleware (e.g., authentication, logging)

│ ├── models/ # Database models (TypeORM, Mongoose, Prisma, etc.)

│ ├── routes/ # Express router definitions

│ ├── services/ # Business logic and reusable service functions

│ ├── helpers/ # Utility functions and helpers

│ ├── validations/ # Request validation schemas (Joi, Zod, etc.)

│ ├── types/ # TypeScript type definitions and interfaces

│ ├── app.ts # Express application setup

│── tests/ # Unit and integration tests

│── .env # Environment variables

│── .gitignore # Ignored files

│── package.json # Project dependencies and scripts  
│── server.ts # Server entry point

│── tsconfig.json # TypeScript configuration

│── README.md # Documentation

## **📌 Detailed Explanation of Each Folder**

### **1. src/config/**

Stores all configuration-related files.

* **Example Files:**
  + db.config.ts – Database connection setup (PostgreSQL, MongoDB, etc.)
  + app.config.ts – Loads and validates environment variables
  + logger.config.ts – Logger configuration (e.g., Winston, Morgan)

### **2. src/controllers/**

Handles request logic and interacts with the service layer.

**Example:** user.controller.ts  
  
import { Request, Response } from "express";

import UserService from "../services/user.service";

class UserController {

async getUsers(req: Request, res: Response) {

const users = await UserService.getAllUsers();

res.json(users);

}

}

export default new UserController();

### **3. src/middlewares/**

Contains Express middleware functions.

* **Examples:**
  + auth.middleware.ts – JWT authentication middleware
  + user.middleware.ts – User handling middleware
  + rateLimit.middleware.ts – Rate limiting middleware

### **4. src/models/**

Defines database models using an ORM like **TypeORM, Prisma, or Mongoose**.

**Example:** user.model.ts (Mongoose)  
  
import mongoose, { Schema, Document } from "mongoose";

interface IUser extends Document {

name: string;

email: string;

password: string;

}

const UserSchema = new Schema<IUser>({

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

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

password: { type: String, required: true }

});

export default mongoose.model<IUser>("User", UserSchema);

### **5. src/routes/**

Contains route definitions using **Express Router**.

**Example:** user.route.ts  
  
import { Router } from "express";

import UserController from "../controllers/user.controller";

const router = Router();

router.get("/", UserController.getUsers);

export default router;

### **6. src/services/**

Encapsulates business logic and database interactions.

**Example:** user.service.ts  
  
import UserModel from "../models/user.model";

class UserService {

async getAllUsers() {

return UserModel.find();

}

}

export default new UserService();

### **7. src/helpers/**

Contains utility/helper functions.

* **Examples:**
  + bcrypt.helper.ts – Password hashing
  + jwt.helper.ts – JWT token handling

### **8. src/validations/**

Handles request body validation using **Joi** or **Zod**.

**Example:** user.validation.ts  
  
import Joi from "joi";

export const userValidationSchema = Joi.object({

name: Joi.string().required(),

email: Joi.string().email().required(),

password: Joi.string().min(6).required(),

});

### **9. src/types/**

Contains TypeScript type definitions.

**Example:** user.type.ts  
t  
export interface User {

id: string;

name: string;

email: string;

}

### **10. src/app.ts**

The main Express application setup.

import express from "express";

import userRoutes from "./routes/user.route";

const app = express();

app.use(express.json());

app.use("/users", userRoutes);

export default app;

### **11. src/server.ts**

The entry point to start the server.

import app from "./src/app";

import dotenv from "dotenv";

dotenv.config();

const PORT = process.env.PORT || 5000;

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

**📌 Best Practices**

✅ **Use TypeScript for Type Safety**✅ **Follow MVC (Model-View-Controller) Architecture**✅ **Separate Business Logic from Controllers**✅ **Use Middleware for Request Processing**✅ **Validate User Input with Joi/Zod**✅ **Use a Logger (Winston, Morgan) for Debugging**✅ **Write Unit Tests & Integration Tests**

## **📌 Recommended Dependencies**

npm install express mongoose dotenv joi jsonwebtoken bcryptjs cors helmet morgan

npm install --save-dev typescript ts-node @types/node @types/express @types/mongoose @types/joi @types/jsonwebtoken @types/bcryptjs

This SOP ensures a **scalable, maintainable, and efficient** Node.js REST API using TypeScript. Let me know if you need further customizations! 🚀