**1.PROJECT OVERVIEW**

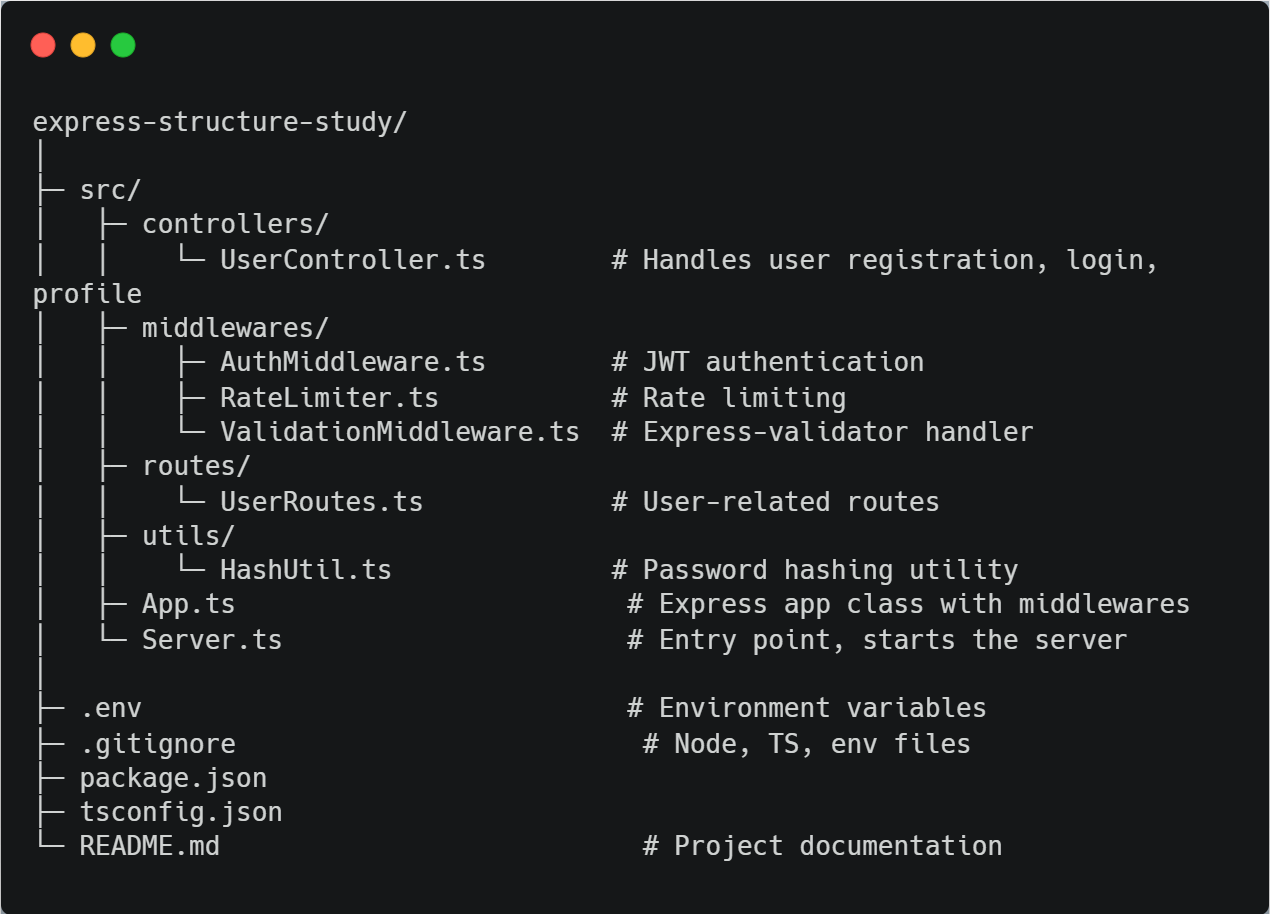
**Name:** Express TypeScript Server Study Project

**Purpose:** Full study of an Express server setup with TypeScript, including security, validation, authentication, and middleware usage.

**Stack:**

* **Backend:** Node.js + Express + TypeScript
* **Security:**
  + helmet → sets HTTP headers
  + cors → enables cross-origin requests
  + express-mongo-sanitize → prevents NoSQL injection
  + express-rate-limit → protects from brute-force attacks
  + compression → response compression
* **Auth:** JWT-based authentication (jsonwebtoken)
* **Password Handling:** bcryptjs
* **Validation & Sanitization:** express-validator
* **Logging:** morgan
* **Environment Variables:** dotenv

**2.FOLDER STRUCTURE**



**3.INSTALLATION**

3.1.Run this command in your project root (where package.json is located):

**Code:**

npm install

* This will automatically install both dependencies (needed for the app to run) and devDependencies (needed for development, like TypeScript, nodemon, etc.).
* After this, a node\_modules folder will be created with all installed packages.

**3.2.INSTALL DEPENDENCIES INDIVIDUALLY (OPTIONAL)**

If you want to install **dependencies** one by one:

**Command:**

npm install express cors helmet compression morgan dotenv bcrypt jsonwebtoken express-validator express-mongo-sanitize express-rate-limit

Then install **devDependencies**:

**Command:**

npm install --save-dev typescript ts-node nodemon @types/node @types/express @types/cors @types/helmet @types/morgan @types/bcrypt @types/jsonwebtoken @types/express-rate-limit

**3.3. VERIFY INSTALLATION**

After installation, you can check if node\_modules exists and all packages are installed:

**Command:**

ls node\_modules

Or check installed versions:

**Command:**

npm list

**4.Stack Overview**

This backend is built with **Node.js, Express, and TypeScript**, with a focus on security, authentication, input validation, and maintainability. Below is a brief explanation of each part of the stack.

**4.1. BACKEND**

**4.1.1. NODE.JS + EXPRESS**

Node.js provides the runtime to execute JavaScript on the server, while Express offers a flexible framework to handle routing, middleware, and API endpoints. Together, they form the backbone of the application, enabling efficient request–response handling.

**4.1.2.TYPESCRIPT**

TypeScript adds static typing on top of JavaScript, which helps catch errors during development and improves maintainability. It makes the project more scalable by enforcing type safety and providing better tooling support.

**4.2. SECURITY**

**4.2.1. HELMET**

Helmet automatically sets various HTTP headers to secure the application. It protects against common attacks like cross-site scripting (XSS), clickjacking, and MIME-type sniffing.

**4.2.2. CORS**

CORS (Cross-Origin Resource Sharing) defines which domains can access the backend. This is essential when the frontend and backend are hosted on different domains, ensuring only trusted clients can interact with the API.

**4.2.3. EXPRESS-MONGO-SANITIZE**

Prevents NoSQL injection attacks by removing suspicious operators (like $ or .) from user inputs before they reach MongoDB queries. This ensures malicious inputs don’t compromise the database.

**4.2.4.** **EXPRESS-RATE-LIMIT**

Limits the number of requests a client can make in a given timeframe. This helps mitigate brute-force login attempts and denial-of-service (DoS) attacks.

**4.2.5.** **COMPRESSION**

Compresses HTTP responses (using Gzip/Deflate) before sending them to the client. This reduces payload size and improves overall performance.

**4.3.** **AUTHENTICATION & PASSWORD HANDLING**

**4.3.1.** **JSONWEBTOKEN (JWT)**

JWTs are used for stateless authentication. After a user logs in, the server issues a token that the client includes in subsequent requests. This avoids storing sessions on the server and makes scaling easier.

**4.3.2.** **BCRYPTJS**

Passwords are hashed using bcrypt before being stored in the database. This ensures that even if the database is compromised, raw passwords remain protected.

**4.4. VALIDATION & SANITIZATION**

**4.4.1.** **EXPRESS-VALIDATOR**

Provides a set of middlewares to validate and sanitize incoming request data. It ensures inputs like emails, passwords, and IDs meet expected formats, reducing the risk of bad or malicious data being processed.

**4.5.** **LOGGING & ENVIRONMENT VARIABLES:**

**4.5.1.** **MORGAN**

A request logger that captures details such as request method, URL, status, and response time. It helps in monitoring and debugging by keeping track of API activity.

**4.5.2.** **DOTENV**

Manages environment variables by loading them from a .env file into process.env. This keeps sensitive information (like database URIs, JWT secrets) secure and out of the codebase.

**5.Module Overview**

**5.1.NODE\_MODULES**

The node\_modules folder is an automatically generated directory that contains **all the packages and dependencies** installed for the Node.js project via **npm** (Node Package Manager) or **yarn**.

**Purpose:**

* Provides the actual code for all **third-party libraries** your project depends on.
* Ensures that your application has all necessary modules like express, cors, helmet, bcrypt, jsonwebtoken, and others.
* Enables Node.js to resolve import or require statements in your project.

**Key Points:**

1. **Automatically managed:**
   * Never manually edit the files inside node\_modules.
   * Managed via package.json and package-lock.json (or yarn.lock) files.
2. **Dependency tree:**
   * Each package may have its own dependencies, which are recursively installed inside node\_modules.
3. **Exclusion from version control:**
   * Usually, node\_modules is **ignored in git** using .gitignore because it can be very large and is reproducible using npm install.
4. **Installation:**
   * Run npm install or yarn to generate node\_modules from package.json.
   * All required packages for the project are installed according to the versions specified in package.json.

**Best Practices:**

* Always include package.json and package-lock.json in your repository to allow others to install dependencies easily.
* Do **not commit node\_modules** to version control.

**Relation to Project:**

* Your Express server, middlewares, controllers, and utilities rely on node\_modules for functionality.
* Examples:
  + express → for building the server
  + cors, helmet, compression → for security and performance
  + bcrypt → for password hashing
  + jsonwebtoken → for authentication
  + express-validator → for input validation

**5.2.PACKAGE.JSON**

The package.json file serves as the **central configuration and metadata file** for the Node.js project. It defines the project’s identity, dependencies, scripts, and other settings required to run and manage the application.

**Purpose:**

* **Project Metadata:** Contains details like project name, version, description, author, and license.
* **Dependency Management:** Lists all libraries required for runtime and development, ensuring consistent setup across different environments.
* **Scripts:** Automates tasks such as running the server in development, starting the production server, or building TypeScript files.
* **Configuration:** Can include settings for tools, testing frameworks, or runtime environments.

**Key Points:**

1. **Metadata:** Provides general information about the project.
2. **Dependencies:** Runtime packages essential for the project to function.
3. **Dev Dependencies:** Packages needed only during development or compilation, such as TypeScript, nodemon, or ts-node.
4. **Scripts:** Predefined commands to simplify project workflows (development server, production start, or build process).
5. **Version Consistency:** Works with package-lock.json to ensure consistent dependency versions for all developers.

**Importance in the Project:**

* Ensures that anyone cloning the repository can install the correct dependencies and run the project.
* Enables automation of repetitive tasks like starting the server or compiling TypeScript.
* Centralizes project information and configuration in a standard format recognized by Node.js and npm.

**5.3.TSCONFIG.JSON**

The tsconfig.json file is the **TypeScript configuration file** that defines how the TypeScript compiler should process the project’s files. It controls compilation options, module resolution, and the output behavior.

**Purpose:**

* Configures **TypeScript compilation** rules for the project.
* Ensures consistent behavior across different environments and developers.
* Allows the use of modern JavaScript features while targeting a specific runtime version.

**Key Points:**

1. **Compiler Options:** Defines how TypeScript should compile files, such as target ECMAScript version, module system, strict type checking, and source maps.
2. **Include / Exclude:** Specifies which files or directories should be compiled or ignored by the TypeScript compiler.
3. **Paths & Aliases:** Allows aliasing for imports to simplify module paths.
4. **Type Checking:** Can enable strict type checking, helping catch errors early during development.
5. **Integration:** Works with tools like ts-node and build scripts to run TypeScript code directly or compile it to JavaScript.

**Importance in the Project:**

* Ensures type safety and reduces runtime errors.
* Standardizes project compilation so all developers and deployment environments behave consistently.
* Enables smooth integration with Node.js, Express, and other TypeScript-based libraries in the project.

**5.4.SERVER.TS**

**Purpose:**  
Server.ts is the **entry point** of the application. Its main role is to **launch the Express server** and make it ready to handle incoming HTTP requests.

**Responsibilities:**

1. **Import App Instance**
   * Imports the configured Express app (App.ts), which includes all **routes, middlewares, and security configurations**.
2. **Set the Port**
   * Reads the port number from **environment variables** (e.g., .env) to allow flexible deployment across environments.
   * Provides a **default port** if none is specified (commonly 5000).
   * Ensures that the server listens on the correct port so clients can access the API endpoints.
3. **Start the Server**
   * Calls Express’s listen method to start the server.
   * Logs a confirmation message indicating the server is running and on which port.

**Benefits of this Structure:**

* **Modularity:** Separates server startup from app configuration.
* **Testability:** Allows testing of the app logic without starting the server.
* **Maintainability:** Clear project structure, easier to scale or modify.

**Optional Enhancements:**

* Graceful shutdown on termination signals.
* Error handling for startup issues.
* HTTPS support for production environments.

**5.5.APP.TS**

**Purpose:**  
App.ts is the core **application setup module**. It configures the Express app with all necessary middlewares, security measures, request parsing, and routes. Essentially, it defines **how the server behaves** for incoming HTTP requests.

**Responsibilities:**

1. **Body Parsing (Request Handling)**
   * Parses incoming JSON and URL-encoded data.
   * Sets request body size limits to prevent **large payload attacks**.
2. **Security Middlewares**
   * **CORS:** Controls cross-origin requests.
   * **Helmet:** Adds security headers to protect against common web vulnerabilities.
   * **Compression:** Reduces response size for faster delivery.
   * **Morgan:** Logs HTTP requests for debugging and monitoring.
3. **Rate Limiting**
   * Uses RateLimiter middleware to prevent **DDoS attacks** or request flooding.
4. **MongoDB Sanitization (Optional)**
   * Sanitizes request inputs to prevent **NoSQL injection attacks**.
   * Note: Directly mutating req.query may crash in newer Node.js versions, so special care is required.
5. **Route Initialization**
   * Registers **module-specific routes**, e.g., /api/users linked to UserRoutes.
   * Ensures **clean separation of concerns** between route definitions and controller logic.

**Benefits of this Structure:**

* **Modularity:** Separates server configuration from server startup (Server.ts).
* **Maintainability:** Easy to add/remove middlewares or routes.
* **Security:** Centralized security configuration for the entire app.
* **Scalability:** Supports additional modules, APIs, or microservices.

**Summary:**  
App.ts is the backbone of your Express application, responsible for **middleware setup, request handling, security enforcement, and route registration**. It ensures that all incoming requests are properly processed and secured before reaching the controllers.

**5.6.USERROUTES.TS**

**Purpose:**  
UserRoutes.ts defines all **HTTP endpoints related to user operations**. It serves as the bridge between incoming requests and the **controller logic**, ensuring that requests are properly validated, authenticated, and authorized before reaching the business logic.

**Responsibilities:**

1. **Public Routes**
   * **POST /register** → Handles user registration.
     + Uses UserController.registerValidation to validate request body.
     + Uses ValidationMiddleware to handle validation results.
     + Calls UserController.register to process the registration.
   * **POST /login** → Handles user login.
     + Validates credentials using UserController.loginValidation.
     + Handles validation errors via ValidationMiddleware.
     + Authenticates user and returns a JWT token.
2. **Protected Routes**
   * **GET /profile** → Retrieves user profile.
     + Requires authentication via AuthMiddleware.authenticate().
     + Calls UserController.getProfile to return user data.
   * **GET /admin** → Admin dashboard access.
     + Requires authentication and admin role: AuthMiddleware.authenticate("admin").
     + Calls UserController.adminDashboard.
3. **Key Features**
   * **Middleware chaining:** Each route applies validation and authentication in sequence.
   * **bind(UserController) usage:** Ensures the correct this context when using class methods.
   * **Separation of concerns:** Routes only define endpoints and middleware; business logic stays in controllers.

**Benefits:**

* Centralized definition of user-related endpoints.
* Ensures security and data validation before reaching the controllers.
* Easy to maintain and extend with new user operations.
* Follows **RESTful principles** for clear API design.

**Summary:**  
UserRoutes.ts is the **entry point for all user API requests**, connecting validation, authentication, and controller logic. It enforces security and correctness while keeping routing logic separate from business logic.

**5.7. RATELIMITER**

The RateLimiter module protects the application from **abuse or excessive requests**. It enforces a limit on the number of requests a single client (identified by their IP address) can make within a defined time period.

**Current Configuration in This Project:**

* **Time window:** 15 minutes
* **Maximum requests per IP:** 100
* **Behavior when limit exceeded:** The server responds with a message asking the client to try again later.

By applying this globally, every client IP is restricted to **no more than 100 requests per 15 minutes**, which helps prevent issues like **DDoS attacks, brute-force login attempts, or server overload**.

**Benefits:**

* Protects the server from excessive traffic by individual clients.
* Ensures fair usage for all users.
* Centralized configuration makes it simple to maintain and adjust limits as needed.

**5.8.AUTHMIDDLEWARE**

The AuthMiddleware module is responsible for **protecting routes** by ensuring that only authenticated users can access them. It works by verifying a **JSON Web Token (JWT)** sent in the request headers.

**Key Features:**

* **Token Verification:** Checks that a valid JWT is present in the Authorization header.
* **Role-Based Access Control:** Optionally restricts access to users with a specific role (e.g., "admin").
* **User Information:** If the token is valid, the decoded user data (username and role) is attached to the request object for use in subsequent route handlers.
* **Error Handling:** Returns appropriate HTTP responses for missing, invalid, or unauthorized tokens.

**Usage in This Project:**

* Applied to **protected routes** like user profile and admin dashboard.
* Can enforce **role-specific access**, preventing non-admin users from accessing admin-only endpoints.

**Benefits:**

* Ensures that only authorized users can access sensitive resources.
* Centralizes authentication logic, making it reusable across multiple routes.
* Supports fine-grained access control via roles.

**5.9.VALIDATIONMIDDLEWARE**

The ValidationMiddleware module is responsible for **checking the validity of incoming request data** before it reaches the controller logic. It works in conjunction with **express-validator**, which defines the rules for each route.

**Key Features:**

* **Error Detection:** Collects validation errors from the request (e.g., missing fields, invalid formats).
* **Standardized Response:** If validation fails, it responds with a 400 Bad Request status and a detailed list of errors.
* **Next Function:** If all validations pass, it calls next() to pass control to the next middleware or controller.

**Usage in This Project:**

* Applied to routes like **user registration** and **login** to ensure that inputs such as username, password, and role meet required criteria.
* Acts as a **pre-controller gatekeeper**, preventing invalid or malformed data from being processed.

**Benefits:**

* Ensures data integrity and reduces potential runtime errors.
* Provides consistent error messages to clients.
* Centralizes validation logic, making it reusable and maintainable.

**5.9. HASHUTIL**

The HashUtil module is responsible for **securely handling user passwords** using hashing. It abstracts the implementation details of password encryption and comparison, ensuring that raw passwords are never stored in plain text.

**Key Features:**

* **Password Hashing:** Converts a plain text password into a secure hashed string before storing it in the database.
* **Password Comparison:** Safely compares a plain text password with a hashed password during login or authentication.
* **Use of Bcrypt:** Utilizes bcrypt, a strong and widely used hashing algorithm that adds salt to prevent attacks such as rainbow table attacks.

**Usage in This Project:**

* Applied during **user registration** to hash passwords before saving them.
* Applied during **user login** to verify that the entered password matches the stored hash.

**Benefits:**

* Enhances security by preventing storage of plain text passwords.
* Protects user accounts even if the database is compromised.
* Encapsulates hashing logic for consistent usage across the project.

**5.10. USERCONTROLLER**

The UserController module handles all **user-related operations** in the application, including registration, login, and access to protected resources. It integrates **validation, password hashing, and authentication** to ensure secure and consistent user management.

**Key Features:**

1. **In-Memory User Storage:**
   * Uses a static array to store users for demonstration purposes.
   * Each user object contains username, hashed password, and role.
2. **Validation Rules:**
   * registerValidation: Ensures that username and password meet minimum requirements and that role (optional) is either "user" or "admin".
   * loginValidation: Ensures that username and password are provided.
   * These rules work with the ValidationMiddleware to prevent invalid requests.
3. **User Registration:**
   * Receives user data from the request.
   * Hashes the password using HashUtil.
   * Stores the new user in the static array.
   * Responds with a success message.
4. **User Login:**
   * Finds the user by username.
   * Compares the submitted password with the hashed password using HashUtil.
   * If valid, generates a JWT token with the username and role.
   * Responds with the token for subsequent authenticated requests.
5. **Protected Routes:**
   * getProfile: Returns user-specific data for authenticated users.
   * adminDashboard: Returns admin-specific data, restricted to users with the "admin" role.
   * Relies on AuthMiddleware for token verification and role-based access.

**Benefits:**

* Provides a centralized location for all user operations.
* Ensures security by hashing passwords and using JWT for authentication.
* Integrates seamlessly with validation and authentication middleware for consistent request handling.
* Demonstrates role-based access control for different endpoints.

**5.11. .ENV**

The .env file stores **environment-specific configuration variables** for the application. Using a separate environment file allows the app to keep sensitive information and configuration **outside the codebase**, making it safer, more flexible, and easier to maintain.

**Current Variables:**

1. **PORT**
   * Defines the port number on which the Express server listens.
   * Example: PORT=5000 means the server will run at http://localhost:5000.
   * Changing this value allows the application to run on different ports without modifying the code.
2. **JWT\_SECRET**
   * Secret key used for signing and verifying **JSON Web Tokens (JWTs)**.
   * Example: JWT\_SECRET=supersecretkey.
   * Keeping this key in the .env ensures that it is not exposed in the source code, enhancing security.

**Usage in Application:**

* Accessed via process.env.PORT and process.env.JWT\_SECRET.
* Provides flexibility for running the app in different environments (development, staging, production) with **different secrets or ports**.

**Best Practices:**

* Do **not commit .env to version control** (use .gitignore) to avoid exposing sensitive information.
* Use descriptive names for environment variables for clarity.
* Optionally, maintain separate .env files for different environments, e.g., .env.development and .env.production.

**5.12.** **.GITIGNORE**

The **.gitignore** file is used to tell Git which files and directories should be ignored and not tracked in the repository. This is important for:

* **Security** → Keeps sensitive files (like .env with API keys and secrets) out of version control.
* **Clean Codebase** → Prevents build artifacts, logs, and temporary files from cluttering the repository.
* **Collaboration** → Ensures that unnecessary local/system files don’t get committed by mistake.

In a **Node.js + TypeScript project**, .gitignore typically excludes things like node\_modules/, build folders (dist/ or build/), log files, and environment variable files.