Implementing a Barebones Multi-Tenant MERN Application with Modern Tooling

Preamble: The Importance of a "Barebones" Foundation
Before delving into the specific implementation details, it is crucial to establish a clear understanding of the "barebones" philosophy guiding this project. The objective is not to construct a feature-replete, market-ready product at this stage. Instead, the primary goal is to lay a robust, scalable, and maintainable architectural foundation. Every component selected and every decision made must reflect this principle, prioritizing correct structure, clean interfaces, and seamless integration over an extensive list of functionalities. This foundational approach ensures that as the application inevitably grows and evolves, it does so upon solid, well-considered ground, minimizing technical debt and facilitating future development efforts.

---

1. Introduction: Visualizing the Barebones MERN Application

- **Overview of the Project Goals and Technology Stack**
  This report outlines the research, design, and implementation strategy for a multi-tenant MERN (MongoDB, Express.js, React, Node.js) application. The core objective is to construct a "barebones" system, emphasizing a strong architectural setup rather than an exhaustive feature set. Key characteristics include a multi-tenant architecture with Role-Based Access Control (RBAC), a frontend built with React, Vite, Shadcn UI, and Lucide Icons, and a backend leveraging Node.js and Express.js. The project will be structured as a monorepo managed with npm workspaces and concurrently. Development and deployment will utilize Docker, with a single root .env file for environment configuration, supplemented by separate frontend and backend configuration modules. Essential backend features include user and tenant management, authentication, API documentation via Swagger UI, and centralized logging with Winston, all built using ECMAScript Modules (ESM).
  The technology stack comprises:
  - **M**ongoDB: NoSQL database for flexible data storage.
  - **E**xpress.js: Backend framework for Node.js.
  - **R**eact: Frontend library for building user interfaces.
  - **N**ode.js: JavaScript runtime environment for the backend.
  - Vite: Frontend tooling for fast development and optimized builds.
  - ESM: Modern JavaScript module system for both frontend and backend.
  - npm Workspaces: For monorepo package management.
  - concurrently: To run multiple development scripts simultaneously.
  - Docker & Docker Compose: For containerization and local service orchestration.
  - Shadcn UI & Lucide Icons: For UI components and iconography.
  - Winston: For centralized backend logging.
  - Swagger UI: For API documentation.

- **High-Level Architecture Diagram (Conceptual)**
  To visualize the project, a conceptual architecture can be described as follows:
  - **Monorepo Root:** The top-level directory containing the entire project. It houses the global package.json for workspace definitions, the root .env file for all environment variables, and a config/ directory for separate frontend and backend configuration modules.
  - **packages/frontend:** This workspace contains the React application, built with Vite. It will include all UI components (Shadcn UI, Lucide Icons), state management logic, and API interaction services.
  - **packages/backend:** This workspace houses the Node.js and Express.js API server. It includes Mongoose for MongoDB interaction, authentication logic, RBAC middleware, user and tenant management services, Winston for logging, and Swagger UI for API documentation.
  - **packages/shared-utils (Optional):** While the "barebones" nature might defer its immediate creation, this workspace could be added later for shared code between frontend and backend, such as TypeScript types/interfaces for API contracts or common validation functions. Its potential existence influences initial API design.
  - **Docker Environment:** A Docker Compose setup will orchestrate the local development environment, running the frontend application, backend API, and a MongoDB instance as separate, interconnected containers.
    The primary interactions within this architecture are:
  1. The Frontend application communicates with the Backend API for data and business logic.
  2. The Backend API interacts with the MongoDB database for data persistence.

This structure promotes a clear separation of concerns. The "barebones" nature of the application, when combined with this monorepo structure, places a strong emphasis on establishing well-defined boundaries and clear API contracts between the frontend and backend packages from the outset. Even with a limited set of initial features, the way these packages are structured and how they communicate will set a crucial precedent for future development. Poorly defined initial communication patterns or data contracts, even in a barebones application, would necessitate significant refactoring as the application scales. Therefore, the conceptual design must not only depict these components as distinct entities but also implicitly highlight the API contract as the primary interface governing their interaction within the monorepo.[1]Furthermore, the decision to adopt a monorepo structure for a "barebones" application suggests a forward-looking perspective, anticipating future growth where shared utilities, type definitions, or even common UI components might become beneficial. The initial architectural setup, while simple, should be designed to facilitate such future expansions without requiring extensive re-engineering. Monorepos are particularly advantageous for code sharing. While a minimal application might not immediately leverage numerous shared packages, the choice of a monorepo indicates an expectation that such sharing will become valuable as the application matures (e.g., for shared validation logic or type definitions ensuring consistency in API contracts). Consequently, the initial setup, though streamlined, should not create barriers to the straightforward addition of a packages/shared-types or packages/shared-components workspace at a later stage. This foresight directly influences the design of initial API contracts and data models, encouraging clarity and maintainability from day one.

---

2. Section 1: Monorepo Foundation with npm and Concurrently

This section details the establishment of the project's foundational structure using npm workspaces and concurrently to manage the frontend and backend packages within a unified monorepo.

- **Setting up the Project Root and package.json for Workspaces**
  The first step is to create the root directory for the monorepo. Inside this root, initialize a package.json file using the command npm init -y. This file will serve as the central configuration point for the entire monorepo.
  The crucial configuration for enabling monorepo capabilities is the workspaces property in this root package.json. This property is an array that defines the paths to the individual packages (or workspaces) within the monorepo. For this project, it will be configured as:
  JSON
  ```
  {
    "name": "barebones-mern-monorepo",
    "private": true,
    "workspaces": [
      "packages/frontend",
      "packages/backend"
    ],
    //... other root configurations
  }
  ```

  This setup is fundamental as it allows npm to manage dependencies across multiple packages efficiently.[1] The structure often involves an apps directory for runnable applications and a packages directory for shared libraries or, as in this case, distinct application components like frontend and backend housed within a packages folder.[3]
  The root package.json will also contain development dependencies that are common across the entire monorepo. This includes tools like concurrently for running multiple services, and potentially global linting tools (e.g., ESLint, Prettier) and testing frameworks. Centralizing these ensures that all packages utilize the same versions of these development tools, promoting consistency and simplifying upgrades.
- **Integrating concurrently for Simultaneous Service Management**
  To streamline the development workflow, concurrently will be used to start and manage the frontend and backend development servers simultaneously with a single command.
  Install concurrently as a development dependency in the root of the monorepo:
  ```
  npm install concurrently --save-dev
  ```
  Then, add a script to the scripts section of the root package.json:
  JSON
  ```
  "scripts": {
  ```

```
  "dev": "concurrently \"npm:dev:frontend\" \"npm:dev:backend\"",
  "dev:frontend": "npm run dev --workspace=frontend",
  "dev:backend": "npm run dev --workspace=backend"
}
```

This dev script uses concurrently to execute the dev scripts defined within the package.json files of the frontend and backend workspaces, respectively. The --workspace flag (or npm run dev -w frontend) is the standard npm way to target scripts in specific workspaces.[1] Each package (frontend and backend) will have its own dev script tailored to its specific needs (e.g., vite for the frontend, nodemon server.mjs for the backend).

- **Structuring packages (e.g., frontend, backend)**
  Within the monorepo root, a packages directory will be created. This directory will house the individual applications:
  - packages/frontend: Contains the React/Vite application.
  - packages/backend: Contains the Node.js/Express API.

  Each of these directories will be an independent npm package with its own package.json file, listing its specific dependencies and scripts. For instance, packages/frontend/package.json will list React, Vite, Shadcn UI, and Lucide Icons as dependencies, while packages/backend/package.json will include Express, Mongoose, Winston, etc.While the "barebones" nature of this project might not immediately necessitate a packages/shared-utils directory for common code like type definitions or validation functions, the monorepo structure inherently supports such additions in the future. Acknowledging this potential allows for a more scalable design, even if this shared package is deferred initially. The organizational pattern of distinguishing between apps (as entry-points) and packages (as dependencies or shared code) is a common and effective practice in monorepos.[3]
- **ESM Configuration Across the Monorepo**
  The entire application will utilize ECMAScript Modules (ESM).
  - **Frontend (React/Vite):** Vite natively supports ESM, so the frontend package will generally work with ESM out of the box.
  - **Backend (Node.js/Express):** The package.json file within packages/backend must include "type": "module". This instructs Node.js to treat .js files in that package as ES modules. Alternatively, .mjs extensions can be used for module files. All import and export statements in the backend code must use ESM syntax (e.g., import express from 'express'; instead of const express = require('express');).

  This consistent use of ESM across both frontend and backend simplifies potential code sharing (if shared-utils is introduced later) and aligns with modern JavaScript development practices.[1]
- **Centralized Environment Configuration**
  A single, centralized approach to environment variable management is a key requirement.
  - Root .env File Strategy:

A single .env file will be located at the project root. This file will store all environment variables required for local development across all parts of the application (frontend, backend, Docker services). Crucially, this .env file must be added to the project's .gitignore file to prevent committing sensitive credentials or configurations to version control.5

- Frontend and Backend Config Modules:
  To manage the access and distribution of these environment variables in a structured way, a config directory will be created at the project root. Inside this directory, two ESM modules will be established:
  - config/frontend.config.mjs: This module will be responsible for exposing environment variables specifically needed by the frontend.
  - config/backend.config.mjs: This module will handle the loading and exposure of environment variables for the backend.

  This approach provides a clean separation and a single source of truth for how each part of the application consumes configuration.[1]
- **Loading and Accessing Environment Variables:**
  - **Backend (Node.js with dotenv):** The dotenv package will be installed as a dependency within the packages/backend workspace (npm install dotenv --workspace=backend). In packages/backend/src/config/backend.config.mjs (assuming source files are in a src subdirectory of the backend package), dotenv will be configured to load the root .env file. Due to the monorepo structure and where the script is run from, path resolution is critical. If scripts are run from the root, dotenv.config({ path: path.resolve(process.cwd(), '.env') }) might work. If run from the package directory, the path might be dotenv.config({ path: path.resolve(process.cwd(), '../../.env') }). A robust way is to define the path relative to the config file itself:
    JavaScript
    ```
    // packages/backend/src/config/backend.config.mjs
    import dotenv from 'dotenv';
    import path from 'path';
    import { fileURLToPath } from 'url';

    const __filename = fileURLToPath(import.meta.url);
    const __dirname = path.dirname(__filename);

    // Adjust path to point to the root.env file
    // Assuming this file is in packages/backend/src/config/
    // and.env is at the monorepo root.
    const rootEnvPath = path.resolve(__dirname, '../../../../.env');
    dotenv.config({ path: rootEnvPath });

    export const MONGODB_URI = process.env.MONGODB_URI;
    ```

export const PORT = process.env.PORT |

| 5000;
export const JWT_SECRET = process.env.JWT_SECRET;
//... other backend variables
~~~
This module then exports the necessary variables in a structured manner.8

* **Frontend (React with Vite):**
Vite handles environment variables differently. Variables prefixed with `VITE_` in an `.env` file (which can be the root `.env` file if Vite is configured to look there, or a separate `.env` in the frontend package directory that sources from the root or is managed by Docker Compose for containerized environments) are exposed to the frontend code via `import.meta.env`.
The `packages/frontend/src/config/frontend.config.mjs` module will access these:
```javascript
// packages/frontend/src/config/frontend.config.mjs
export const API_BASE_URL = import.meta.env.VITE_API_BASE_URL |

| 'http://localhost:5000/api';
export const TENANT_ID_HEADER = 'x-tenant-id';
//... other frontend variables
~~~
The root .env file should contain these VITE_ prefixed variables for local development. For Dockerized environments, these will be passed through docker-compose.yml. This aligns with how build tools typically embed environment variables into the frontend bundle.7

The use of npm workspaces inherently simplifies the management of dependencies across the `frontend` and `backend` packages. By hoisting common dependencies to the root `node_modules` folder, duplication is reduced, and version consistency is more easily maintained.[1, 2, 3] `concurrently` further enhances the development experience by enabling the simultaneous execution of multiple package scripts (like starting development servers for both frontend and backend) with a single command from the root.[4]

However, this combination of ESM, npm workspaces, and a single root `.env` file introduces a nuanced challenge regarding how individual packages access this central `.env` file. Path resolution becomes a critical consideration. For the backend, `dotenv` needs to be pointed correctly to the root `.env` file, often requiring navigation up several directory levels (e.g., `../../.env` or more, depending on the file structure and current working directory). Using `import.meta.url` and `path.resolve` as shown above provides a more robust way to calculate this path relative to the config file itself, rather than relying on `process.cwd()` which can vary. For the frontend, build tools like Vite manage `.env` loading, but ensuring they pick up variables from the root `.env` (or that these variables are correctly passed during the build process, especially in Docker) is key. The `config/frontend.config.mjs` and `config/backend.config.mjs` modules serve as crucial intermediaries, abstracting this path logic and providing a clean interface for the respective applications to consume configuration variables.[1, 5, 8]

Adopting ESM across the entire monorepo from the project's inception aligns with modern JavaScript standards and can improve aspects like tree-shaking and asynchronous module loading. However, this choice necessitates vigilance in ensuring that all selected tools and libraries—including Express middleware, Mongoose, Winston, Swagger, and any testing frameworks—offer stable and straightforward ESM support. This might influence library version choices or require specific configurations (e.g., for Jest or Nodemon when used with ESM projects).[3] Some libraries might have different import patterns (named vs. default imports) or require specific entry points when consumed as ES modules. For instance, some older libraries might not have full ESM compatibility, potentially requiring wrappers or alternative packages.[10] Therefore, during the implementation phase, careful attention must be dedicated to how each dependency is imported and utilized within an ESM context to preempt compatibility issues.

To provide a clear overview of all environment variables used in the project, the following table is instrumental:

Table: Environment Variable Overview

Variable Name	Description	Used By	Example Value	Loaded Via
PORT	Backend server port	Backend	5000	root .env -> backend.config.mjs
MONGODB_URI	MongoDB connection string	Backend, Docker	mongodb://localhost:27017/mern_barebones	root .env -> backend.config.mjs
JWT_SECRET	Secret key for signing JWTs	Backend	your-very-strong-jwt-secret	root .env -> backend.config.mjs
JWT_EXPIRES_IN	JWT expiration time (e.g., 1h, 7d)	Backend	1h	root .env -> backend.config.mjs
VITE_API_BASE_URL	Base URL for frontend to call backend API	Frontend, Docker	http://localhost:5000/api	root .env -> Vite build / frontend.config.mj

				s
NODE_ENV	Application environment (development/production)	Backend, Docker	development	System / docker-compose.yml
LOG_LEVEL	Winston logging level (info, debug, error)	Backend	info	root .env -> backend.config.mjs (logger)
DOCKER_FRONTEND_PORT	Exposed port for frontend Docker container	Docker	3000	root .env -> docker-compose.yml
DOCKER_BACKEND_PORT	Exposed port for backend Docker container	Docker	5000	root .env -> docker-compose.yml
DOCKER_MONGO_PORT	Exposed port for MongoDB Docker container	Docker	27017	root .env -> docker-compose.yml

This table serves as a centralized reference, crucial for managing configurations in a system where a single `.env` file influences multiple components and Dockerized services. It aids in debugging, setup, and onboarding.

---

3. Section 2: Backend Implementation (Node.js, Express.js, MongoDB)

This section details the construction of the server-side application, focusing on API development, data management using MongoDB with Mongoose, implementing multi-tenancy, Role-Based Access Control (RBAC), authentication, API documentation with Swagger, and centralized logging with Winston. All backend code will adhere to ESM standards.

- **Core Express.js Setup with ESM**
  The backend will be built using Express.js. The setup involves initializing an Express application using ESM syntax:
  JavaScript
  ```
  // packages/backend/src/server.mjs (or index.mjs)
  import express from 'express';
  import cors from 'cors';
  import { PORT } from './config/backend.config.mjs';
  import connectDB from './config/db.mjs';
  import logger from './utils/logger.mjs'; // Winston logger
  ```

```
    // Import route handlers
    import authRoutes from './routes/auth.routes.mjs';
    import userRoutes from './routes/user.routes.mjs';
    import tenantRoutes from './routes/tenant.routes.mjs';
    // Import error handling and tenant context middleware
    import { tenantContextMiddleware } from './middleware/tenantContext.middleware.mjs';
    import { errorHandler } from './middleware/errorHandler.middleware.mjs';

    const app = express();

    // Connect to MongoDB
    connectDB();

    // Core Middleware
    app.use(cors({
      origin: process.env.FRONTEND_URL |

| 'http://localhost:3000', // Configure allowed origins
credentials: true // Important for HttpOnly cookies
}));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));



// Request Logging Middleware (Winston integrated with Morgan or custom)
// Example: app.use(morgan('combined', { stream: logger.stream })); // See Winston section

// Tenant Context Middleware (must come after auth if tenant is derived from user or JWT)
app.use(tenantContextMiddleware);

// API Routes
app.use('/api/auth', authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/tenants', tenantRoutes);

// Swagger UI Setup (details later)

// Centralized Error Handling Middleware (must be last)
app.use(errorHandler);

app.listen(PORT, () => {
  logger.info(`Backend server running on port ${PORT} in ${process.env.NODE_ENV} mode`);
```

~~~
});
```
~~~

The structure will involve a main router in `server.mjs` (or `app.mjs`/`index.mjs`) that delegates to modular routers for different resources (e.g., `authRoutes`, `userRoutes`, `tenantRoutes`). This modular approach is a best practice for organizing routes in Express applications.[11]

- **MongoDB and Mongoose Integration**
  Mongoose will be used as the Object Data Modeling (ODM) library to interact with MongoDB. The connection to MongoDB will be established using the connection string sourced from packages/backend/src/config/backend.config.mjs.
  Schema Design for Multi-Tenancy and RBAC:
  The database schema is critical for implementing multi-tenancy and RBAC effectively. The following Mongoose models will be defined, adapted from best practices and specific project requirements 12:
  - **User Model (packages/backend/src/models/user.model.mjs):**
    JavaScript
    ```javascript
    import mongoose from 'mongoose';
    import bcrypt from 'bcryptjs';

    const userSchema = new mongoose.Schema({
      username: { type: String, required: true, unique: true, trim: true },
      email: { type: String, required: true, unique: true, trim: true, lowercase: true },
      password: { type: String, required: true, select: false }, // select: false to exclude by default
      isSystemAdmin: { type: Boolean, default: false },
    }, { timestamps: true });

    // Pre-save hook to hash password
    userSchema.pre('save', async function(next) {
      if (!this.isModified('password')) return next();
      const salt = await bcrypt.genSalt(10);
      this.password = await bcrypt.hash(this.password, salt);
      next();
    });

    // Method to compare password
    userSchema.methods.comparePassword = async function(candidatePassword) {
      if (!this.password) return false; // User might not have password if isSystemAdmin and created differently
      return bcrypt.compare(candidatePassword, this.password);
    };
    ```

```
const User = mongoose.model('User', userSchema);
export default User;
```

Key fields include username, email, password (hashed using bcryptjs), and isSystemAdmin. The isSystemAdmin flag is essential for distinguishing global administrators who can operate across tenants.[12]

- **Tenant Model (packages/backend/src/models/tenant.model.mjs):**
  JavaScript
  ```
  import mongoose from 'mongoose';

  const tenantSchema = new mongoose.Schema({
    name: { type: String, required: true, unique: true },
    slug: { type: String, required: true, unique: true, trim: true, lowercase: true }, // For URL identification
    createdBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
    settings: { type: Object, default: {} }, // For tenant-specific configurations
  }, { timestamps: true });

  const Tenant = mongoose.model('Tenant', tenantSchema);
  export default Tenant;
  ```

  This model represents an individual tenant (e.g., an organization). It includes a name, a URL-friendly slug, and a reference to the User who created it.[12]
- **Membership Model (packages/backend/src/models/membership.model.mjs):**
  JavaScript
  ```
  import mongoose from 'mongoose';

  const membershipSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
    tenant: { type: mongoose.Schema.Types.ObjectId, ref: 'Tenant', required: true },
    role: {
      type: String,
      enum:, // System Admin is a global role on User model
      required: true,
    },
  }, { timestamps: true });

  // Compound unique index to ensure a user has only one role per tenant
  membershipSchema.index({ user: 1, tenant: 1 }, { unique: true });

  const Membership = mongoose.model('Membership', membershipSchema);
  export default Membership;
  ```

This model is pivotal for linking users to tenants and defining their role *within that specific tenant*. The role field uses an enum to restrict values to 'Tenant Admin', 'Author', or 'User'. The 'System Admin' role is managed via the isSystemAdmin flag on the User model, as it's a global role, not tenant-specific.[12]

These Mongoose schemas are fundamental [13] and draw heavily from the structure proposed in [12], which aligns well with the project's multi-tenancy and RBAC requirements.To clearly define permissions, an RBAC Permission Matrix is essential:**Table: RBAC Permission Matrix**

Role	**Resource**	**Create**	**Read Own**	**Read Any (Tenant)**	**Read All (System)**	**Update Own**	**Update Any (Tenant)**	**Update All (System)**	**Delete Own**	**Delete Any (Tenant)**	**Delete All (System)**	**Manage Tenant Users**	**Manage Tenant Settings**	**Manage All Tenants**
System Admin	User	Yes	N/A	N/A	Yes	N/A	N/A	Yes	N/A	N/A	Yes	Yes (any tenant)	Yes (any tenant)	Yes
System Admin	Tenant	Yes	N/A	N/A	Yes	N/A	N/A	Yes	N/A	N/A	Yes	N/A	N/A	Yes
System Admin	Content (within tenant)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	N/A	N/A	N/A
Tenant Admin	User (within own tenant)	Yes	N/A	Yes	No	N/A	Yes	No	N/A	Yes	No	Yes	Yes	No
Tenant Admin	Tenant (own)	No	Yes	Yes	No	Yes	Yes	No	No	No	No	N/A	Yes	No
Tenant Admin	Content (within own tenan	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	N/A	N/A	No

	t)													
Author	Content (within own tenant)	Yes	Yes	No	No	Yes	No	No	Yes	No	No	No	No	No
User	Content (within own tenant)	No	Yes	No	No	No	No	No	No	No	No	No	No	No
User	User (own profile)	N/A	Yes	N/A	N/A	Yes	N/A	N/A	N/A	N/A	N/A	No	No	No

This matrix serves as the blueprint for implementing authorization logic.

- **Multi-Tenancy Implementation**
  The chosen strategy is a **Single Database, Shared Schema with tenantId**. In this model, data for all tenants coexists within the same MongoDB collections, but each relevant document includes a tenantId field to distinguish ownership and scope queries.[15] This is a common and practical approach for SaaS applications, balancing data isolation with operational simplicity.[15]
  - **Tenant Identification:** The primary method for identifying the current tenant context for an API request will be through a custom HTTP header, x-tenant-id. This is straightforward for Single Page Applications (SPAs) to implement.[15] The frontend will be responsible for sending this header when a user is operating within a specific tenant's context.
  - Express.js Middleware for Tenant Context (packages/backend/src/middleware/tenantContext.middleware.mjs):
    A middleware will be created to extract the tenantId from the x-tenant-id request header. It will validate the tenantId (e.g., by checking if a tenant with that ID exists in the Tenant collection) and, if valid, attach the tenantId (and potentially the full tenant object fetched from the database) to the req object (e.g., req.tenantId, req.tenant). This makes the tenant context readily available to subsequent middleware and route handlers.

JavaScript
```javascript
import Tenant from '../models/tenant.model.mjs';
import mongoose from 'mongoose';
import logger from '../utils/logger.mjs';

export const tenantContextMiddleware = async (req, res, next) => {
  const tenantIdHeader = req.headers['x-tenant-id'];

  if (tenantIdHeader) {
    if (!mongoose.Types.ObjectId.isValid(tenantIdHeader)) {
      logger.warn(`Invalid tenantId format in header: ${tenantIdHeader}`);
      return res.status(400).json({ message: 'Invalid tenant ID format.' });
    }
    try {
      const tenant = await Tenant.findById(tenantIdHeader);
      if (!tenant) {
        logger.warn(`Tenant not found for ID: ${tenantIdHeader}`);
        return res.status(404).json({ message: 'Tenant not found.' });
      }
      req.tenant = tenant; // Attach full tenant object
      req.tenantId = tenant._id.toString(); // Attach tenantId string
      logger.debug(`Tenant context set for tenantId: ${req.tenantId}`);
    } catch (error) {
      logger.error(`Error fetching tenant by ID ${tenantIdHeader}:
${error.message}`);
      return res.status(500).json({ message: 'Error processing tenant information.'
});
    }
  }
  next();
};
```

This middleware is crucial for enforcing data isolation at the application level.[16]

- **Role-Based Access Control (RBAC)**
  RBAC will be implemented based on the roles defined: System Admin (global), Tenant Admin, Author, and User (all tenant-specific).
  - **Express.js Middleware for Authorization (packages/backend/src/middleware/auth.middleware.mjs - combined with JWT verification):** A flexible authorization middleware will be developed. This middleware will be responsible for:
    1. Verifying the JWT (see Authentication section below).
    2. Populating req.user with authenticated user details (including isSystemAdmin).

3. If a tenantId is present in req (set by tenantContextMiddleware), fetching the user's role within that specific tenant from the Membership collection and adding it to req.user.currentTenantRole.
4. Providing a function, say checkPermission(requiredRolesOrPermissions), that route handlers can use to verify access.

JavaScript

```
// packages/backend/src/middleware/auth.middleware.mjs
import jwt from 'jsonwebtoken';
import { JWT_SECRET } from '../config/backend.config.mjs';
import User from '../models/user.model.mjs';
import Membership from '../models/membership.model.mjs';
import logger from '../utils/logger.mjs';

export const protect = async (req, res, next) => {
  let token;
  if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
    token = req.headers.authorization.split(' ');
  } else if (req.cookies && req.cookies.token) { // Check HttpOnly cookie
    token = req.cookies.token;
  }

  if (!token) {
    return res.status(401).json({ message: 'Not authorized, no token' });
  }

  try {
    const decoded = jwt.verify(token, JWT_SECRET);
    const user = await User.findById(decoded.id).select('-password'); // Exclude password

    if (!user) {
      return res.status(401).json({ message: 'Not authorized, user not found' });
    }
    req.user = user; // Attach full user object (without password)

    // If tenant context is set by tenantContextMiddleware, fetch role in that tenant
    if (req.tenantId) {
      const membership = await Membership.findOne({ user: user._id, tenant: req.tenantId });
      if (membership) {
        req.user.currentTenantRole = membership.role;
      } else if (!user.isSystemAdmin) {
        // If not a system admin and no membership, they shouldn't access tenant-specific
resources
        // This check might be more nuanced depending on the route
         logger.warn(`User ${user._id} has no membership in tenant ${req.tenantId} but is trying
to access a tenant-scoped route.`);
        // return res.status(403).json({ message: 'Forbidden: No membership in this tenant.' });
      }
    }
    next();
  } catch (error) {
    logger.error(`Token verification failed: ${error.message}`);
    return res.status(401).json({ message: 'Not authorized, token failed' });
  }
};

export const authorize = (roles =) => {
  // roles can be an array of strings:,, ['Author', 'User']
  // or specific permissions like ['manage:users', 'read:content'] if a more granular system is
built later
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({ message: 'Not authorized' });
    }

    const userIsSystemAdmin = req.user.isSystemAdmin;
    const roleInCurrentTenant = req.user.currentTenantRole;

    // System Admin has universal access if 'System Admin' is in roles
    if (roles.includes('System Admin') && userIsSystemAdmin) {
      return next();
    }

    // If not a system admin check or if system admin role is not sufficient for the specific tenant
action
    // Check tenant-specific role
    if (roleInCurrentTenant && roles.includes(roleInCurrentTenant)) {
      return next();
    }

    // Fallback for System Admin if the required role is tenant-specific but SA should have
access
    // This logic might need refinement based on how SA interacts with tenant data
    if (userIsSystemAdmin && req.tenantId && (roles.includes('Tenant Admin') |

| roles.includes('Author') |
| roles.includes('User'))) {
logger.debug(System Admin ${req.user._id} accessing tenant ${req.tenantId} resource
requiring roles: ${roles.join(', ')});
return next(); // System admin can act as any role within a tenant context
}
```

```
      logger.warn(`Authorization failed for user ${req.user._id}. Required roles: <span
class="math-inline">\{roles\.join\(', '\)\}\. User roles\:
isSystemAdmin\=</span>{userIsSystemAdmin}, currentTenantRole=${roleInCurrentTenant}`);
      return res.status(403).json({ message: 'Forbidden: You do not have the required role.' });
    };
  };
```

This middleware structure allows for flexible permission checks on routes.[26, 27]

- User Management API (CRUD Operations)
  Endpoints: POST /api/users, GET /api/users, GET /api/users/:id, PUT /api/users/:id, DELETE /api/users/:id. These will be protected by the protect and authorize middleware.
  - **System Admin (authorize()):**
    - Can create any user, potentially assign isSystemAdmin.
    - Can list all users across all tenants (requires careful query, possibly populating memberships).
    - Can view, update (including roles within tenants via Membership model), and delete any user.
    - Controller logic will check req.user.isSystemAdmin. If true, queries are not scoped by tenantId.
    - **Snippet Integration:** [12] discuss System Admin capabilities.
  - **Tenant Admin (authorize()):**
    - Can create users *within their own tenant*. When creating a user, a Membership record linking the new user to the req.tenantId with a role ('Author' or 'User') must also be created.
    - Can list, view, update (e.g., roles like 'Author', 'User'), and delete users *only within their req.tenantId*. All Mongoose queries must be scoped: User.find({ _id: { $in: userIdsFromTenantMemberships } }) or by modifying Membership records.
    - Controller logic will use req.tenantId to scope all operations.
    - **Snippet Integration:** [12] explicitly states "Tenant Admins can manage all users...within their tenant." [12] cover Mongoose queries scoped by tenantId.
- Tenant Management API (CRUD Operations)
  Endpoints: POST /api/tenants, GET /api/tenants, GET /api/tenants/:id, PUT /api/tenants/:id, DELETE /api/tenants/:id.
  - **System Admin Only (authorize()):** Only System Admins can perform CRUD operations on tenants.
    - When a System Admin creates a tenant, they must also designate an initial Tenant Admin for that tenant by creating a User (if not existing) and a Membership record.
    - **Snippet Integration:**.[87]
- **Authentication**

- **JWT-based Authentication Flow:**
    - Login (POST /api/auth/login): Validates credentials (email, password). If valid, generates a JWT.
    - Register (POST /api/auth/register): Creates a new user. For this barebones app, initial registration might not automatically create a tenant or assign to one; this could be a System Admin or Tenant Admin function.
- **Token Generation (Payload):** The JWT payload is critical for efficient RBAC. It should contain:
    - userId: The ID of the authenticated user.
    - isSystemAdmin: Boolean flag.
    - If the user selects a tenant to operate within (e.g., after login or via a tenant switcher on the frontend), the token can be re-issued or a separate mechanism can set the active tenant context. For simplicity in a barebones app, if a user is part of only one tenant, currentTenantId and roleInCurrentTenant (from their Membership record for that tenant) can be included. If they belong to multiple, the frontend might need to specify the tenant via x-tenant-id header, and the backend then verifies membership and role for that tenant on each request. For this report, we'll assume the x-tenant-id header sets the context, and the auth.middleware.mjs enriches req.user with currentTenantRole.
    - **Snippet Integration:**.[29]
- **Secure Token Storage (HttpOnly Cookies):** The JWT will be sent to the client in an HttpOnly cookie to mitigate XSS risks. This is a security best practice.[28]

  JavaScript
  ```
  // Example in login controller
  res.cookie('token', token, {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production', // Only send over HTTPS in production
    sameSite: 'strict', // Mitigate CSRF
    maxAge: 3600000 // 1 hour, e.g.
  });
  ```
- **Express.js Middleware for Token Verification and User Hydration:** This is handled by the protect middleware detailed above. It extracts the JWT from the HttpOnly cookie (or Authorization header as a fallback), verifies it, and if valid, decodes the payload to attach user information to req.user.

- **API Documentation with Swagger UI**
    - Setup with swagger-jsdoc and swagger-ui-express for ESM:
      The backend will use swagger-jsdoc to generate an OpenAPI specification from JSDoc comments in the route files, and swagger-ui-express to serve the interactive Swagger UI.
      JavaScript

```
// packages/backend/src/server.mjs (additions)
import swaggerUi from 'swagger-ui-express';
import swaggerJSDoc from 'swagger-jsdoc';

const swaggerOptions = {
  swaggerDefinition: {
    openapi: '3.0.0',
    info: {
      title: 'Barebones MERN API',
      version: '1.0.0',
      description: 'API documentation for the Barebones MERN Application',
    },
    servers:,
    components: {
      securitySchemes: {
        bearerAuth: { // Matches the name used in route security
          type: 'http',
          scheme: 'bearer',
          bearerFormat: 'JWT',
        },
      },
    },
    security: [{ bearerAuth: }], // Apply globally if all/most routes are protected
  },
  apis: ['./src/routes/*.mjs'], // Path to the API docs
};
const swaggerSpec = swaggerJSDoc(swaggerOptions);
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec));
```

This setup defines the OpenAPI 3.0 spec, including a security scheme for JWT Bearer authentication.[30]

- **Documenting JWT-protected routes:** JSDoc comments in route files will specify security requirements.

JavaScript
```
// Example in packages/backend/src/routes/user.routes.mjs
/**
 * @swagger
 * /users:
 *   get:
 *     summary: Get all users (System Admin) or users in tenant (Tenant Admin)
 *     tags: [Users]
 *     security:
 *       - bearerAuth: # Indicates this route requires JWT Bearer token
 *     parameters:
 *       - in: header
 *         name: x-tenant-id
 *         schema:
 *           type: string
 *         required: false # Required for Tenant Admin, optional for System Admin
 *         description: The ID of the tenant to filter users by (for Tenant Admins)
 *     responses:
 *       200:
 *         description: A list of users.
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 $ref: '#/components/schemas/User' // Define User schema elsewhere
 *       401:
 *         description: Unauthorized
 *       403:
 *         description: Forbidden
 */
router.get('/', protect, authorize(), userController.getAllUsers);
```

This ensures API documentation clearly indicates which routes are protected and how to authenticate.[30]

- **Centralized Logging with Winston**
  - **Creating a Reusable Logger Module (ESM):** A dedicated module, packages/backend/src/utils/logger.mjs, will configure Winston.
    JavaScript
    ```
    // packages/backend/src/utils/logger.mjs
    import winston from 'winston';
    const { combine, timestamp, json, colorize, printf, errors } = winston.format;

    const consoleFormat = printf(({ level, message, timestamp, stack }) => {
      return `${timestamp} ${level}: ${stack |
    ```

```
| message}`;
});
```

```
const logger = winston.createLogger({
  level: process.env.LOG_LEVEL |
```

```
| 'info',
format: combine(
colorize(),
timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
errors({ stack: true }), // Log stack traces
consoleFormat // For development, human-readable
),
transports: [new winston.transports.Console()],
exitOnError: false,
});



  if (process.env.NODE_ENV === 'production') {
    logger.clear().add(new winston.transports.Console({
      format: combine(
        timestamp(),
        errors({ stack: true }),
        json() // JSON format for production
      ),
    }));
    // In a real production scenario, add transport for centralized logging service here
    // e.g., new winston.transports.Http({ host: 'log-server', port: '3000' })
  }

  // Create a stream object with a 'write' function that will be used by Morgan
  logger.stream = {
      write: (message) => {
          logger.http(message.trim()); // Use 'http' level for request logs
      },
  };

  export default logger;
```
This module sets up different logging behavior for development (colored console output) and production (JSON output, suitable for ingestion by centralized logging systems).[33, 34]

* **Integrating with Express.js for Request and Error Logging:**
  Morgan can be used for HTTP request logging, configured to pipe its output to Winston.
  ```javascript
  // packages/backend/src/server.mjs (additions)
  import morgan from 'morgan'; // npm install morgan
  ```

```
   //...
   // Morgan middleware for request logging - pipe to Winston
   // Use 'combined' for Apache standard logging, or a custom format
   // The stream option directs Morgan's output to Winston's http level
   app.use(morgan('combined', { stream: logger.stream }));


   // Centralized Error Handling Middleware (modified to use logger)
   // This should be the last middleware
   app.use((err, req, res, next) => {
     logger.error(`${err.status |

| 500} - ${err.message} - ${req.originalUrl} - ${req.method} - ${req.ip}`, {
error: err, // Log the full error object
stack: err.stack,
user: req.user? req.user.id : 'Guest',
tenantId: req.tenantId |
| 'N/A'
});
res.status(err.status |
| 500).json({
message: err.message |
| 'An unexpected error occurred.',
// Optionally include stack in development
stack: process.env.NODE_ENV === 'development'? err.stack : undefined,
});
});
```
This setup ensures that both standard HTTP requests and application errors are logged consistently.34

The choice of a "single database, shared schema with `tenantId`" is a pragmatic and widely adopted strategy for multi-tenancy in MERN applications. It balances the need for data isolation with operational simplicity, which is particularly suitable for a "barebones" project that aims for a solid foundation.[16] Coupled with JWTs stored in HttpOnly cookies, this provides a secure authentication mechanism.[28]

Implementing RBAC effectively in this multi-tenant context requires careful consideration of how roles are scoped. The `Membership` model, linking `User`, `Tenant`, and `Role`, is central to this.[12] Authorization middleware must perform a two-fold check: first, verify the user's identity via the JWT, and second, determine their specific role *within the context of the current tenant*. The `currentTenantId` (obtained from the `x-tenant-id` header and processed by `tenantContextMiddleware`) is used to query the `Membership` data or to validate against claims in an enriched JWT. To optimize performance and reduce database lookups on every protected route, the JWT payload design is critical. Including `userId`, `isSystemAdmin`, `currentTenantId` (if a tenant context is active), and `roleInCurrentTenant` in the JWT can significantly improve efficiency.[29] This makes the JWT a temporary, verifiable cache for the user's contextual role.

The "System Admin" role introduces an important distinction in the RBAC logic. This role operates above and across tenant boundaries. Consequently, the authorization middleware needs a specific conditional check: if `req.user.isSystemAdmin` (a claim derived from the User model and included in the JWT) is true, tenant-specific scoping for certain operations (like managing tenants themselves or viewing all users across all tenants) should be bypassed. For actions *within* a specific tenant (e.g., modifying a user who belongs to Tenant A), a System Admin might either act with the effective permissions of a Tenant Admin for that specific tenant or possess distinct global permissions that apply. This duality must be clearly defined in the RBAC Permission Matrix and meticulously implemented in the authorization middleware. The `Membership.role` field would typically not store "System Admin" as a tenant-specific role; instead, the `User.isSystemAdmin` flag governs global administrative capabilities, while `Membership.role` defines roles *within* the confines of a particular tenant.

---

4. Section 3: Frontend Implementation (React, Vite, Shadcn UI, Lucide Icons)

This section details the development of the client-side application using React with Vite, incorporating Shadcn UI for components and Lucide Icons for iconography. The focus is on creating a user interface that is responsive, thematically consistent, and adapts to different user roles.

- **Setting up React with Vite (ESM)**
  A new React project will be initialized within the packages/frontend directory using Vite, which natively supports ESM and offers a fast development experience.
  npm create vite@latest frontend -- --template react (or react-ts if TypeScript is preferred for the frontend, though the request implies a JavaScript MERN stack, Vite handles TS well).
  The basic project structure will follow Vite's conventions, typically including an index.html, src/main.jsx (or .tsx), and src/App.jsx.
- **Integrating Shadcn UI**
  Shadcn UI is chosen for its unique approach to component building, providing unstyled, accessible components that can be fully customized with Tailwind CSS.
  - Initial Setup and Adding Components:
    The integration process involves initializing Tailwind CSS in the packages/frontend project, configuring tailwind.config.js, postcss.config.js, and creating a components.json file as per Shadcn UI's official documentation. Components are

added using the Shadcn UI CLI (e.g., npx shadcn-ui@latest add button card input tabs). This command copies the component source code directly into the project, typically under a components/ui directory, granting full ownership and control over the component code.35

- Customization and Theming for a Consistent Design:
  A consistent design theme will be achieved by:
  1. **Tailwind Configuration:** Modifying packages/frontend/tailwind.config.js to define the application's primary color palette, typography (fonts), spacing, border radius, and other design tokens. These configurations will be used by Tailwind CSS utility classes that style the Shadcn UI components.[37]
  2. **CSS Variables:** Utilizing CSS variables (custom properties) in a global CSS file (e.g., packages/frontend/src/index.css or globals.css). Shadcn UI components are designed to be themed using CSS variables for aspects like background, foreground, primary/secondary colors, accents, borders, and radius. These variables are defined for both light and dark modes.[36]
     Example in index.css:
     CSS
     ```css
     @tailwind base;
     @tailwind components;
     @tailwind utilities;

     @layer base {
      :root {
        --background: 0 0% 100%;
        --foreground: 222.2 84% 4.9%;
        /*... other light theme variables... */
        --radius: 0.5rem;
      }
      .dark {
        --background: 222.2 84% 4.9%;
        --foreground: 210 40% 98%;
        /*... other dark theme variables... */
      }
     }
     ```
  3. **Dark Mode:** Implementing dark mode support using Tailwind's darkMode: 'class' strategy. A theme toggler component (e.g., using a button with a Lucide icon) will switch between light and dark themes by adding or removing the dark class from the <html> or <body> element.[37] Online tools like TweakCN [39] or the Shadcn UI Customizer [40] can assist in visualizing and generating theme configurations.
- Best Practices for Component Organization:

    Shadcn UI components copied via the CLI are typically placed in packages/frontend/src/components/ui. Application-specific components, built using these UI primitives or for specific features, should be organized in separate directories like packages/frontend/src/components/shared (for reusable across features) or packages/frontend/src/features/[featureName]/components.36

- **Using Lucide Icons**
  Lucide Icons will provide a comprehensive and tree-shakable set of SVG icons.
  - **Installation:** npm install lucide-react within the packages/frontend workspace.[41]
  - **Importing and Usage:** Specific icons are imported as React components, ensuring only used icons are bundled.
    JavaScript
    ```
    import { Mail, Lock, User } from 'lucide-react';
    //...
    <Button>
      <Mail className="mr-2 h-4 w-4" /> Login with Email
    </Button>
    ```
  - **Customization:** Props like size, color, and strokeWidth can be passed to customize the icons' appearance.[42]
- **State Management (Context API)**
  For the "barebones" nature of this application, React's built-in Context API is a suitable choice for managing global application state. This includes:
  - **Authentication State:** User object, authentication status (e.g., isAuthenticated), JWT token (if managed client-side, though HttpOnly cookies are preferred for the token itself, the frontend might store user info decoded from it).
  - **User Information:** Current user's details, including roles (isSystemAdmin, currentTenantRole).
  - **Tenant Context:** Current tenantId if the user is operating within a specific tenant.
    Separate contexts (e.g., AuthContext, UserContext, TenantContext) will be created and provided at the application root or relevant sub-trees.
- **Implementing Role-Specific Landing Pages/Dashboards**
  The application must present different views and functionalities based on the authenticated user's role and their current tenant context.
  - Conditional Rendering:
    React components will use conditional rendering logic based on the user's role (System Admin, Tenant Admin, Author, User) and, if applicable, their active tenant. This information will be available from the global state (Context API).
    - **System Admin Dashboard:** Will feature components for managing tenants (Tenant CRUD) and users across all tenants (User CRUD).
    - **Tenant Admin Dashboard:** Will display components for managing users (User CRUD within their tenant) and potentially content or authors specific to their tenant.
    - **Author Dashboard:** Will provide an interface for creating and managing

        content within their assigned tenant.
    - **User Dashboard:** Will offer a basic view of content relevant to them within their tenant and profile management. .[44]
  - Navigation Guards/Private Routes:
    React Router will be used for client-side navigation. Private routes will be implemented to protect certain parts of the application. These routes will check for authentication status and potentially user roles/permissions before rendering the target component.
    JavaScript

```
// Example PrivateRoute component
import { Navigate, Outlet } from 'react-router-dom';
import { useAuth } from './contexts/AuthContext'; // Example auth context

const PrivateRoute = ({ allowedRoles }) => {
  const { isAuthenticated, user } = useAuth();

  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }

  // Check for System Admin first
  if (allowedRoles.includes('System Admin') && user.isSystemAdmin) {
    return <Outlet />;
  }

  // Check for tenant-specific roles
  if (user.currentTenantRole && allowedRoles.includes(user.currentTenantRole)) {
    return <Outlet />;
  }

  // If user is System Admin and the route is for a tenant role, allow access
  // (assuming System Admin can impersonate or has full access within any tenant)
  if (user.isSystemAdmin && (allowedRoles.includes('Tenant Admin') |
```

```
| allowedRoles.includes('Author') |
| allowedRoles.includes('User'))) {
return <Outlet />;
}
```

```
    return <Navigate to="/unauthorized" replace />; // Or to a default dashboard
  };
```

```
Unauthenticated users will be redirected to a login page, and users attempting to access resources without the necessary permissions will be redirected to an "unauthorized" page or their default role-appropriate dashboard.

- **API Integration**
  The frontend will communicate with the backend API using fetch or a library like axios.
  - An API client module can be created to centralize API call logic, including setting base URLs from frontend.config.mjs and automatically attaching the JWT (if managed client-side for non-HttpOnly scenarios, or handling credentials for HttpOnly cookies) and the x-tenant-id header.
    JavaScript
    ```
    // Example apiClient.js
    import axios from 'axios';
    import { API_BASE_URL, TENANT_ID_HEADER } from './config/frontend.config.mjs';

    const apiClient = axios.create({
      baseURL: API_BASE_URL,
      withCredentials: true, // Important for sending/receiving HttpOnly cookies
    });

    apiClient.interceptors.request.use(config => {
      // Get current tenantId from global state (e.g., TenantContext)
      const currentTenantId = /* logic to get currentTenantId from state */;
      if (currentTenantId) {
        config.headers = currentTenantId;
      }
      return config;
    });

    export default apiClient;
    ```
  - Standard data fetching patterns will be used, including handling loading states (e.g., displaying spinners using Shadcn's Spinner or a custom loader) and error states (e.g., displaying error messages using Shadcn's Alert component).
  - The frontend must correctly handle the authentication state, potentially storing user information and roles received after login in the global context. The x-tenant-id header must be included in API calls when the user is operating within a specific tenant's context.[16]

Shadcn UI's philosophy of providing unstyled, copy-pasteable components gives developers complete control over their appearance and behavior, which is ideal for implementing a highly consistent design theme.[35] This, combined with Lucide Icons' tree-shakable nature [42], makes for an efficient and customizable UI toolkit.The establishment of a "consistent design theme"
    ```

using Shadcn UI is heavily reliant on the meticulous initial setup of tailwind.config.js and the global CSS file where CSS variables for theming are defined. Since Shadcn components are copied directly into the components/ui directory, they become part of the project's codebase. While this offers unparalleled customization, it also means that significant changes to the core theme definitions (like renaming CSS variables or altering the fundamental color roles) might require manual updates to these copied components if their internal styling relies on the old definitions. Therefore, careful upfront planning of the theme (colors, typography, spacing, radius) is crucial to minimize rework later.[36]For role-specific dashboards within a multi-tenant application, the frontend's state management must robustly handle not just the authenticated user's identity and global roles (like System Admin) but also their *current tenant context*. This context includes the activeTenantId and the user's specific role *within that active tenant* (e.g., Tenant Admin, Author, User). This is critical because users can potentially belong to multiple tenants with different roles in each.[12] The frontend might need to provide a mechanism for users to switch between their tenant contexts if they are members of more than one. The UI elements rendered, the actions available, and the data fetched via API calls (which will include the x-tenant-id header) are all dictated by this active tenant context and the user's role within it. A global state managed by the Context API, holding userId, isSystemAdmin, activeTenantId, and roleInActiveTenant, becomes indispensable. This state would be populated upon successful login and updated whenever the user switches their active tenant context.

---

5. Section 4: Dockerizing the MERN Application

Containerization with Docker and Docker Compose is essential for creating consistent and reproducible environments for development, testing, and eventual deployment. This section outlines the strategy for Dockerizing the MERN monorepo.

- **Crafting Multi-Stage Dockerfiles**
  Multi-stage Dockerfiles are a best practice for creating optimized, smaller, and more secure production images. They achieve this by separating the build environment (which may contain many development dependencies and tools) from the final runtime environment, which only includes the necessary application code and runtime dependencies.[48]
  - Optimized Dockerfile for the React Frontend (packages/frontend/Dockerfile): This Dockerfile will use a multi-stage build:
    - **Build Stage:**
      1. Start from an official Node.js image (e.g., node:18-alpine or node:20-alpine for a smaller base).
      2. Set the working directory (e.g., /app/frontend).
      3. Copy the root package.json and package-lock.json to leverage Docker's layer caching for dependencies that haven't changed across the monorepo. Also copy the frontend's specific package.json.
      4. Run npm install --workspace=frontend (or npm ci --workspace=frontend) to install only frontend dependencies, respecting the monorepo structure. This step requires careful handling of the package-lock.json as npm workspaces place it at the root.[50] The package-lock.json from the root should be copied to the

        context where npm ci for the workspace is run. A common pattern is to copy the entire monorepo context or structure the Dockerfile to correctly access the root lockfile while installing workspace-specific dependencies.
        5. Copy the frontend source code (e.g., COPY./packages/frontend.).
        6. Run the Vite build command (e.g., npm run build). This will generate static assets in a dist folder.
    - **Serve Stage:**
        1. Start from a lightweight web server image (e.g., nginx:alpine).
        2. Copy the built static assets from the dist folder of the build stage into Nginx's HTML directory (e.g., /usr/share/nginx/html).
        3. Copy a custom Nginx configuration file (nginx.conf) to handle client-side routing (e.g., redirecting all 404s to index.html) and potentially serve gzipped assets.
        4. Expose the Nginx port (e.g., 80). [48]
  - Optimized Dockerfile for the Node.js/Express Backend (packages/backend/Dockerfile):
    This Dockerfile will also use a multi-stage build:
    - **Build/Dependencies Stage:**
        1. Start from an official Node.js image (e.g., node:18-alpine or node:20-alpine).
        2. Set the working directory (e.g., /app/backend).
        3. Copy root package.json, package-lock.json, and backend's package.json.
        4. Run npm ci --workspace=backend --omit=dev to install only production dependencies for the backend, again ensuring the root package-lock.json is accessible and used.
        5. Copy the backend source code (e.g., COPY./packages/backend/src./src).
    - **Run Stage:**
        1. Start from a slim Node.js image (e.g., node:18-alpine-slim or node:20-alpine-slim).
        2. Set the working directory.
        3. Copy the node_modules and source code from the build stage.
        4. Set NODE_ENV=production.
        5. Expose the backend application port (e.g., 5000).
        6. Define the CMD to run the ESM application (e.g., CMD ["node", "src/server.mjs"]). [50]
- **docker-compose.yml for Local Development**
  A docker-compose.yml file at the root of the monorepo will define and orchestrate the services for local development.
  - **Defining frontend, backend, and mongo services:**
    YAML

```
version: '3.8'

services:
  frontend:
    build:
      context:. # Root of the monorepo
      dockerfile:./packages/frontend/Dockerfile
      args: # For passing build-time env vars from.env
        VITE_API_BASE_URL: ${VITE_API_BASE_URL_DEV:-http://localhost:5000/api}
    ports:
      - "${DOCKER_FRONTEND_PORT:-3000}:80" # Map host port from.env to
container's Nginx port 80
    volumes:
      -./packages/frontend/src:/app/frontend/src # Hot reloading for src
      # Note: For Vite HMR with Docker, additional config might be needed in
vite.config.js
    depends_on:
      - backend
    networks:
      - mern-app-network
    env_file:
      -.env # For runtime Nginx env vars, if any

  backend:
    build:
      context:.
      dockerfile:./packages/backend/Dockerfile
    ports:
      - "${DOCKER_BACKEND_PORT:-5000}:${PORT:-5000}" # Map host port
from.env to container's app port
    volumes:
      -./packages/backend/src:/app/backend/src # Hot reloading for src
      -./config:/app/backend/config # Mount shared config
      - /app/backend/node_modules # Prevent host node_modules from overwriting
container's
    depends_on:
      - mongo
    networks:
      - mern-app-network
    env_file:
      -.env # Load variables from root.env file

  mongo:
    image: mongo:latest
    ports:
      - "${DOCKER_MONGO_PORT:-27017}:27017"
    volumes:
      - mongo-data:/data/db # Persistent data for MongoDB
    networks:
      - mern-app-network
    env_file: # Though mongo image uses its own env vars like
MONGO_INITDB_ROOT_USERNAME
      -.env

networks:
  mern-app-network:
    driver: bridge

volumes:
  mongo-data:
    driver: local
```

52

- Managing Build Contexts for Monorepo Structure:
  The context for both frontend and backend services in docker-compose.yml is set to . (the root of the monorepo). The dockerfile property then specifies the path to the respective Dockerfile relative to this root context (e.g., dockerfile:./packages/frontend/Dockerfile). This setup ensures that Docker has access to the entire monorepo structure during the build, which is crucial for handling the root package-lock.json and potentially shared files.50
- Injecting Environment Variables from the Root .env File:
  The env_file: -.env directive in each service definition within docker-compose.yml instructs Docker Compose to load environment variables from the .env file located in the project root. These variables become available to the services at runtime. For the backend service, the MONGODB_URI should be configured to use the Docker internal network hostname for the MongoDB service (e.g., mongodb://mongo:27017/yourdbname, where mongo is the service name defined in docker-compose.yml).52
- Network Configuration:
  A custom bridge network (mern-app-network) is defined to allow the services (frontend, backend, mongo) to communicate with each other using their service names as hostnames.52

The monorepo structure, particularly with npm workspaces, introduces specific considerations for Dockerization. The primary challenge revolves around dependency installation (npm ci) and the location of package.json and package-lock.json files. Npm workspaces typically create a single package-lock.json at the monorepo root and hoist dependencies to a root node_modules folder.[1] When building a Docker image for an individual package (e.g., frontend or backend), the Docker build context usually only includes that package's directory. However, npm ci requires both the package's package.json and the (root) package-lock.json to ensure consistent and verifiable dependency installation. The solution involves setting the Docker build context in docker-compose.yml to the monorepo root. Then, within each Dockerfile, carefully COPY the root package-lock.json, the specific package's package.json, and then run npm ci --workspace=<packageName>. Npm is intelligent enough to only install dependencies relevant to that workspace, using the root lockfile for version integrity.[50]Furthermore, managing environment variables from a single root .env file in a docker-compose setup for a monorepo requires distinguishing between build-time arguments (ARGs) and runtime environment variables (ENV). While env_file in docker-compose.yml primarily sets runtime environment variables for the containers [52], some frontend variables (like VITE_API_BASE_URL) often need to be available during the frontend's build process to be embedded into the static assets.[7] If these build-time variables are also defined in the root .env file, they need to be explicitly passed as build arguments to the frontend service's Dockerfile. This can be achieved using the args sub-property under the build key in the docker-compose.yml service definition. For example:YAML

```yaml
# In docker-compose.yml
services:
  frontend:
    build:
      context:.
      dockerfile:./packages/frontend/Dockerfile
      args:
        VITE_API_BASE_URL: ${VITE_API_BASE_URL_DEV} # Reads from.env or uses default
```

And in the frontend Dockerfile:Dockerfile

```dockerfile
# In packages/frontend/Dockerfile
ARG VITE_API_BASE_URL
ENV VITE_API_BASE_URL=${VITE_API_BASE_URL}
#... later, during npm run build, Vite will pick up VITE_API_BASE_URL
```

Backend services, on the other hand, typically consume environment variables at runtime (e.g., process.env.MONGODB_URI), which are directly supplied by the env_file directive. This dual approach ensures that both build-time and runtime configurations are correctly sourced from the single root .env file.

---

6. Section 5: Requirements, User Stories, and Cursor AI IDE Integration

This section translates the project's objectives into concrete requirements and user stories. It also details how the Cursor AI IDE can be effectively utilized throughout the development lifecycle to accelerate tasks, from initial setup to code generation and debugging.

- **Project Requirements Summary**
  The core requirements for this barebones MERN application are:
  1. **Technology Stack:** MongoDB, Express.js, React, Node.js (MERN).
  2. **Modularity:** ECMAScript Modules (ESM) for both frontend and backend.
  3. **Monorepo:** Structure using npm workspaces, managed with concurrently for simultaneous development server execution.
  4. **Environment Configuration:** Single root .env file, with separate

config/frontend.config.mjs and config/backend.config.mjs modules.

5. **Frontend UI:** Lucide Icons for iconography, Shadcn UI for React components, ensuring a consistent design theme.
6. **Multi-Tenancy:** Backend support for multiple tenants (Single Database, Shared Schema with tenantId).
7. **Role-Based Access Control (RBAC):** Four defined roles: System Admin (global), Tenant Admin (tenant-scoped), Author (tenant-scoped), User (tenant-scoped).
8. **User Management:** CRUD operations for users, respecting RBAC rules (System Admin manages all, Tenant Admin manages users within their tenant).
9. **Tenant Management:** CRUD operations for tenants, restricted to System Admins.
10. **User Interface:** Role-specific landing pages or dashboards displayed upon login.
11. **Authentication:** JWT-based authentication, with tokens preferably stored in HttpOnly cookies.
12. **API Documentation:** Swagger UI for documenting backend APIs.
13. **Logging:** Centralized backend logging using Winston.
14. **Containerization:** Local development and service hosting using Docker and Docker Compose.
15. **Version Control & IDE:** GitHub for version control, Cursor AI IDE for development assistance.
16. **Scope:** The application is "barebones," focusing on establishing the core architecture and specified features correctly.

- **Generating User Stories with Cursor AI**
  User stories will define the application's functionality from the perspective of its users. Cursor AI can assist in generating these stories based on the defined roles and features.[64]
  - **Roles for User Stories:**
    - System Admin
    - Tenant Admin
    - Author
    - User
  - **Key Features for User Stories:**
    - Authentication (Login, Registration - if applicable for self-sign-up, Logout)
    - Tenant Management (CRUD by System Admin)
    - User Management (CRUD by System Admin globally, CRUD by Tenant Admin within their tenant)
    - Role Assignment (System Admin assigns Tenant Admins, Tenant Admin assigns Authors/Users)
    - Dashboard Access (Role-specific views)
    - (Placeholder for content-related actions by Author/User, e.g., "View articles")
  - **Example Prompts for Cursor AI to Generate User Stories:**
    - "Given a multi-tenant MERN application with roles: System Admin, Tenant Admin, Author, User. Generate user stories for the **System Admin** focusing

on tenant lifecycle management (create, read, update, delete tenants) and assigning the initial Tenant Admin to a new tenant."

- "For the same application, generate user stories for a **Tenant Admin** who needs to manage users (invite/create, view, update roles to Author/User, deactivate/delete) exclusively within their own tenant."
- "Generate user stories for an **Author** who needs to create, edit, and view their own articles within their assigned tenant."
- "Generate user stories for a **User** focusing on logging in, viewing their role-specific dashboard, and viewing articles within their tenant."

The AI can help structure these stories in the standard "As a [type of user], I want [an action] so that [a benefit/value]" format, and also suggest acceptance criteria.[64]**Table: User Stories for Key Features**

User Story ID	**As a (Role)**	**I want to...**	**So that...**	**Acceptance Criteria (Example)**
SA-T-001	System Admin	create a new tenant with a unique name and assign an initial Tenant Admin	I can onboard new organizations to the platform.	- Tenant name must be unique. <br/> - An existing user must be assignable as Tenant Admin. <br/> - Tenant Admin receives appropriate permissions for the new tenant.
SA-T-002	System Admin	view a list of all tenants with basic details	I can monitor and manage all organizations on the platform.	- List shows tenant name, creation date, Tenant Admin. <br/> - Pagination if many tenants exist.
SA-U-001	System Admin	view a list of all users across all tenants	I can oversee all user accounts in the system.	- List shows username, email, system admin status, tenant memberships/roles. <br/> - Ability to filter/search users.
SA-U-002	System Admin	create a new system administrator	I can delegate system-wide administrative	- New user is created with isSystemAdmin

		account	tasks.	flag set to true.
TA-U-001	Tenant Admin	invite and add new users (Authors or Users) to my tenant	I can manage my organization's members and their access.	- Can specify user email and role (Author/User). <br/> - New user is created if they don't exist globally, or existing user is added to tenant. <br/> - User is linked to my tenant with the specified role. <br/> - User cannot be assigned 'Tenant Admin' or 'System Admin' role by Tenant Admin.
TA-U-002	Tenant Admin	view a list of all users within my tenant	I can see who belongs to my organization and their roles.	- List only shows users belonging to my tenant. <br/> - User roles within my tenant are displayed.
TA-D-001	Tenant Admin	log in and see a dashboard specific to my tenant management tasks	I can efficiently manage my tenant's users and settings.	- Dashboard shows user count, quick links to add users, etc.
AU-C-001	Author	log in and access a dashboard to create and manage my content	I can contribute articles/posts to my tenant's space.	- Dashboard shows options to create new content. <br/> - Can only see/edit content they created within their tenant.
USR-L-001	User	log in with my credentials	I can access the application's	- Successful login redirects to

			features relevant to my role and tenant.	role-specific dashboard. <br/> - Failed login shows an error message.
USR-D-001	User	view my personalized dashboard after logging in	I can access information and features relevant to me.	- Dashboard content is filtered based on my role and tenant.

- **Leveraging Cursor AI IDE for Development**
  Cursor AI IDE is a powerful tool that can assist throughout the MERN monorepo development process.[67]
  - **Project Setup and Monorepo Navigation in Cursor:**
    - Open the entire monorepo root directory in Cursor. Cursor's file explorer and search capabilities will facilitate navigation between the packages/frontend, packages/backend, and config directories.
    - For large monorepos, managing AI context is key. Use @-mentions to specify relevant files or symbols when prompting. Create .cursor/rules/*.mdc files (Project Rules) to provide persistent, scoped guidance to the AI about the project structure, technologies used in different packages, and coding conventions.[70] For instance, a root-level rule could describe the overall monorepo layout and the purpose of each package, while nested rules within packages/frontend/.cursor/rules/ could specify React/Shadcn best practices.
  - **Code Generation:**
    - **Generating React Components (Shadcn UI, Lucide Icons):**
      - **Prompt Example:** "Generate a React functional component named TenantList.jsx that displays a list of tenants using Shadcn UI's Table, TableRow, TableCell, TableHead, TableHeader, and TableBody components. Each row should show tenant name and creation date. Include a 'Manage Users' button using Shadcn Button with a Lucide Users icon for each tenant. The component should accept tenants array as a prop."
      - Provide context by @-mentioning existing Shadcn UI components in components/ui or relevant type definitions. Cursor can be guided to use these established patterns.[73] A .cursorrule like the one in [75] can enforce Shadcn UI best practices.
    - **Generating Express.js Routes and Controllers (ESM):**
      - **Prompt Example:** "Create an Express.js ESM controller function createTenant in tenant.controller.mjs. It should handle a POST request, take name and adminEmail from the request body. It needs to create a new Tenant using the Mongoose Tenant model and assign the user with adminEmail as 'Tenant Admin' via the Membership

model. Ensure proper async/await, try/catch error handling, and log actions using the imported Winston logger. @-mention TenantModel @-mention MembershipModel @-mention UserModel @-mention logger."
        - Cursor can generate route definitions, controller logic, and basic service layer interactions.[11] An Express.js specific .cursorrule can enforce conventions.[11]
    - **Assistance with Mongoose Schemas:**
        - **Prompt Example:** "Generate a Mongoose schema for 'Membership' linking 'User' and 'Tenant' models. It should include a 'role' field as an enum with values 'Tenant Admin', 'Author', 'User', and ensure a user has a unique role per tenant. Use ESM syntax. @-mention UserSchema @-mention TenantSchema."
- **Debugging Strategies for Frontend and Backend in a Monorepo:**
    - **AI Chat for Errors:** Copy error messages from the browser console (frontend) or terminal (backend) directly into Cursor's chat. Ask "What causes this error and how can I fix it in the context of @path/to/problematic/file.mjs?"
    - **Iterative Fixing:** Cursor's features like "loop on errors" or "iterate on lints" can automatically attempt to fix issues based on linter feedback or compiler errors.[67]
    - **Contextual Debugging:** Select a block of problematic code, use Ctrl+K (or Cmd+K) and ask "Explain this code" or "Find potential bugs in this selected code."
    - **Log Analysis:** If backend logs (Winston) indicate an issue, paste relevant log snippets into Cursor and ask for an interpretation or debugging suggestions. The strategy of "write tests first, then code, then run tests and update code until tests pass" is highly effective with AI, as is adding temporary logs and feeding the output back to Cursor for analysis.[79]
    - For monorepo debugging, clearly specifying which package (frontend or backend) the error pertains to is crucial when prompting Cursor.[80]
- **Assistance with Dockerfile and docker-compose.yml Creation:**
    - **Prompt Example (Dockerfile):** "Generate a multi-stage Dockerfile for the backend service located in packages/backend. It's a Node.js Express ESM application. Stage 1 should install production dependencies using npm ci --workspace=backend --omit=dev (ensure root package-lock.json is used). Stage 2 should be a slim Node image, copy built artifacts and node_modules, and run node src/server.mjs. @-mention./package.json @-mention./packages/backend/package.json."
    - **Prompt Example (docker-compose):** "Create a docker-compose.yml for a MERN monorepo. Define three services: frontend (build context '.', dockerfile packages/frontend/Dockerfile), backend (build context '.',

dockerfile packages/backend/Dockerfile), and mongo (official mongo image). frontend depends on backend, backend depends on mongo. Use the root .env file for environment variables for all services. Map ports appropriately. Create a bridge network mern-app-network."
        - Cursor can generate initial versions of these files based on project structure and requirements.[82]
    - Visualizing Project Architecture and Data Flow (Conceptual Description):
      While Cursor AI doesn't generate graphical diagrams, it can help articulate the architecture and data flows textually.
        - **Prompt Example:** "Based on the Mongoose schemas for User, Tenant, and Membership (@-mention these files), and the Express route structure in packages/backend/src/routes/, describe the data flow and relationships involved when a Tenant Admin creates a new User within their tenant."
        - "Explain the overall architecture of this MERN monorepo, detailing how the frontend, backend, and config packages interact, and how environment variables are managed."
        - Cursor's ability to analyze the codebase, guided by rules and specific file references, can produce these descriptions.[83]
    - Utilizing .cursorrules for MERN development:
      Create project-specific rules in .cursor/rules/*.mdc files to enforce coding standards and architectural patterns.
        - Example Rule (e.g., mern-backend.mdc):
          Markdown
          ```
          ---
          description: "Guidelines for MERN backend development with Express, Mongoose, and ESM."
          globs: ["packages/backend/src/**/*.mjs"]
          alwaysApply: true
          ---
          # Backend Development Rules
          - Always use ESM syntax (import/export).
          - Controllers should only handle request/response logic and call service functions for business logic.
          - Service functions should encapsulate Mongoose queries and business operations.
          - Use async/await for all asynchronous operations.
          - All routes must have appropriate JWT protection and RBAC authorization middleware.
          - Log important actions and errors using the Winston logger (`@-mention packages/backend/src/utils/logger.mjs`).
          - Mongoose models should be defined in `packages/backend/src/models/`.
          ```

Such rules help ensure AI-generated code aligns with project conventions.[72]**Table: Cursor AI**

Prompts for Key Tasks

Development Task	Example Cursor AI Prompt	Key Context/Files to Provide
Create User Mongoose Schema	"Generate a Mongoose schema for 'User' with fields: username (String, required, unique), email (String, required, unique, lowercase), password (String, required, select: false), isSystemAdmin (Boolean, default: false). Include timestamps and a pre-save hook for password hashing using bcryptjs. Use ESM syntax."	
Generate Login React Component	"Create a React functional component LoginForm.jsx using Shadcn UI components: Card for the container, CardHeader, CardTitle, CardContent, CardFooter. Include Input fields for email and password (use Lucide Mail and Lock icons), and a Button for submission. Implement basic state handling for form fields and an onSubmit handler. @-mention packages/frontend/src/components/ui/button.jsx @-mention packages/frontend/src/components/ui/input.jsx @-mention packages/frontend/src/components/ui/card.jsx"	packages/frontend/src/components/ui/ (Shadcn components), Lucide icon import examples.
Create Backend Tenant CRUD Controller	"Generate an Express.js ESM controller in tenant.controller.mjs with CRUD functions (createTenant, getTenants, getTenantById, updateTenant, deleteTenant) for the Tenant Mongoose model. All functions should be async and include try/catch blocks for error handling,	Tenant Mongoose model, auth.middleware.mjs, logger.mjs.

	logging actions with Winston. Ensure System Admin authorization is checked before allowing these operations. @-mention packages/backend/src/models/tenant.model.mjs @-mention packages/backend/src/utils/logger.mjs @-mention packages/backend/src/middleware/auth.middleware.mjs"	
Dockerfile for Backend	"Generate a multi-stage Dockerfile for the Node.js/Express ESM backend in packages/backend/. Stage 1 (builder): use node:20-alpine, copy root package-lock.json and packages/backend/package.json, run npm ci --workspace=backend --omit=dev, copy backend source. Stage 2 (runner): use node:20-alpine-slim, copy artifacts from builder, set NODE_ENV=production, expose backend port, CMD ['node', 'src/server.mjs']. @-mention./package.json @-mention./packages/backend/package.json"	Root and backend package.json.
Debug Backend Error	"I'm getting this error in my Express backend when trying to create a user: [Paste Error Log Here]. The relevant files are @packages/backend/src/controllers/user.controller.mjs and @packages/backend/src/services/user.service.mjs. Can you help identify the cause and suggest a fix?"	Error log, relevant controller and service files.

The effectiveness of Cursor AI, especially in a nuanced project like a MERN monorepo with multi-tenancy and RBAC, is directly proportional to the clarity and context provided in the prompts. Features like `@-mentioning` specific files or symbols, selecting relevant code blocks before invoking AI actions (Ctrl+K or Cmd+K), and crafting detailed prompts that explicitly state constraints (e.g., "using ESM syntax," "for a multi-tenant scenario where data is scoped by `tenantId`," "ensure Shadcn UI `Button` and `Input` are used") are vital for guiding the AI to produce relevant and correct code.[73]

While Cursor AI can assist in generating user stories from high-level feature descriptions and defined roles [65], the intrinsic complexity of multi-tenancy and RBAC rules means that the core domain logic and precise access control definitions must first be clearly articulated by the developer. The AI can then help structure these into formal user stories and acceptance criteria. The process is collaborative: the developer defines the critical business rules (e.g., "A Tenant Admin can only manage users within their own tenant, and cannot assign System Admin roles"), and the AI helps translate these into standard agile artifacts, ensuring all facets of the requirement are considered.[12, 47] This prevents the AI from generating overly generic stories that might miss these critical, domain-specific nuances.

---

7. Conclusion and Next Steps

- **Recap of the Implemented Barebones MERN Application**
  This report has outlined a comprehensive strategy for implementing a "barebones" MERN application, emphasizing a robust architectural foundation. Key decisions include:
  - **Monorepo with npm Workspaces and concurrently:** Facilitates organized code management, shared dependencies, and streamlined development workflows for the frontend and backend packages.[1]
  - **ESM Across the Stack:** Modernizes the codebase, enabling better tree-shaking and module management.[3]
  - **Centralized Configuration:** A single root .env file coupled with dedicated frontend and backend configuration modules provides a clear and manageable approach to environment variables.[1]
  - **Multi-Tenancy (Single Database, Shared Schema):** A practical and scalable approach for isolating tenant data using a tenantId field, with tenant context managed via x-tenant-id headers and backend middleware.[16]
  - **Role-Based Access Control (RBAC):** Clearly defined roles (System Admin, Tenant Admin, Author, User) with permissions managed through Mongoose schemas (User, Tenant, Membership) and enforced by Express.js middleware.[12]
  - **JWT Authentication with HttpOnly Cookies:** Secure authentication mechanism

protecting against XSS, with JWTs carrying essential user and role information.[28]
    - **Shadcn UI and Lucide Icons:** Provides a highly customizable and developer-controlled UI component library and a comprehensive icon set for a consistent frontend design theme.[35]
    - **Swagger UI and Winston:** Ensures comprehensive API documentation and robust, centralized backend logging, respectively.[30]
    - **Docker and Docker Compose:** Enables consistent development and deployment environments through containerization, with optimized multi-stage builds.[48]

  The "barebones" philosophy adopted ensures that while the feature set is minimal, the underlying architecture is sound, scalable, and adheres to modern best practices, preparing the application for future growth.
- **Potential Areas for Future Expansion**
  With this solid foundation in place, the application is well-positioned for future enhancements:
    - **Complex Feature Development:** Building out more sophisticated business logic and features on top of the established multi-tenant and RBAC framework.
    - **Advanced Authentication:** Implementing a full refresh token flow for more persistent user sessions and potentially integrating OAuth2 providers.
    - **State Management:** If frontend complexity grows, migrating from Context API to a more robust state management library like Redux Toolkit or Zustand.
    - **CI/CD Pipelines:** Setting up automated build, test, and deployment pipelines using GitHub Actions or similar tools.
    - **Tenant Customization:** Expanding tenant-specific settings and UI theme customizations.
    - **Granular Permissions:** Evolving the RBAC model to include more fine-grained permissions beyond basic roles, potentially at the resource-instance level.
    - **Real-time Features:** Integrating WebSockets for features like notifications or collaborative editing.
    - **Full-Text Search:** Implementing search capabilities using Elasticsearch or MongoDB's built-in text search.
    - **Background Jobs/Queues:** Using tools like BullMQ or RabbitMQ for handling asynchronous tasks.
    - **Enhanced Monitoring and Alerting:** Integrating with more comprehensive monitoring solutions beyond basic logging.
    - **Scalability Enhancements:** Implementing database read replicas, caching strategies (e.g., Redis), and load balancing as the application scales.

This report provides a detailed blueprint for initiating the development of the barebones MERN application, leveraging modern tools and best practices to create a system that is both functional and prepared for future complexities.

Works cited

1. Node.js handbook | moonrepo, accessed May 11, 2025,

https://moonrepo.dev/docs/guides/javascript/node-handbook
2. Monorepo and its setup for MERN stack - DEV Community, accessed May 11, 2025, https://dev.to/trex777/monorepo-and-its-setup-for-mern-stack-3l66
3. nathanb/sandbox-ts-monorepo: A super basic monorepo ... - GitHub, accessed May 11, 2025, https://github.com/nathanb/sandbox-ts-monorepo
4. Simple Monorepo Setup With React.js And Express.js · dusanstam ..., accessed May 11, 2025, https://dusanstam.com/posts/react-express-monorepo
5. Storing .env files outside of the project root (monorepo) : r/learnpython - Reddit, accessed May 11, 2025, https://www.reddit.com/r/learnpython/comments/1ihr136/storing_env_files_outside_of_the_project_root/
6. Shared configuration in monorepos - Package Management and Workspaces with Yarn for Projects | StudyRaid, accessed May 11, 2025, https://app.studyraid.com/en/read/13159/436741/shared-configuration-in-monorepos
7. Making environment variables accessible in front-end containers ..., accessed May 11, 2025, https://developers.redhat.com/blog/making-environment-variables-accessible-in-front-end-containers
8. How to use environment variables from a .env file in Node.js, accessed May 11, 2025, https://geshan.com.np/blog/2024/11/nodejs-dotenv/
9. Environment Variables - webpack, accessed May 11, 2025, https://webpack.js.org/guides/environment-variables/
10. Configuration Files · node-config/node-config Wiki · GitHub, accessed May 11, 2025, https://github.com/node-config/node-config/wiki/Configuration-Files
11. Express.js Cursor Rules for AI - Playbooks, accessed May 11, 2025, https://playbooks.com/rules/expressjs
12. Implement Multi-Tenancy Role-Based Access Control (RBAC) in ..., accessed May 11, 2025, https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb
13. Mongoose v8.14.2: Schemas, accessed May 11, 2025, https://mongoosejs.com/docs/guide.html
14. How do I setup a member roles for a Users schema using mongoose - Stack Overflow, accessed May 11, 2025, https://stackoverflow.com/questions/52765169/how-do-i-setup-a-member-roles-for-a-users-schema-using-mongoose
15. Implementing Multi-Tenant Architecture in Web Applications - GUVI, accessed May 11, 2025, https://www.guvi.in/blog/multi-tenant-architecture-in-web-applications/
16. How to Build a Multi-Tenant SaaS Application with the MERN Stack - DEV Community, accessed May 11, 2025, https://dev.to/nadim_ch0wdhury/how-to-build-a-multi-tenant-saas-application-with-the-mern-stack-4bl9
17. Multi-Tenant: Database Per Tenant or Shared? - CodeOpinion, accessed May 11, 2025, https://codeopinion.com/multi-tenant-database-per-tenant-or-shared/
18. Build a Multi-Tenant Architecture in MongoDB | GeeksforGeeks, accessed May 11,

2025, https://www.geeksforgeeks.org/build-a-multi-tenant-architecture-in-mongodb/
19. Guide to Building Multi-Tenant Node.js Applications: Architectures, Security, and Real-World Examples - Context Neutral, accessed May 11, 2025, https://www.contextneutral.com/guide-building-applications-architectures-security/
20. What is the best way to do multi-tenant with node/express/mysql? - Stack Overflow, accessed May 11, 2025, https://stackoverflow.com/questions/61308695/what-is-the-best-way-to-do-multi-tenant-with-node-express-mysql
21. React Multi-Tenant Web Application Development - Appilian, accessed May 11, 2025, https://appilian.com/react-multi-tenant-web-application-development/
22. Implementing Multi-tenancy with Mongoose - MoldStud, accessed May 11, 2025, https://moldstud.com/articles/p-implementing-multi-tenancy-with-mongoose
23. aliakseiherman/mern-multitenancy: multi-tenant MERN architecture - GitHub, accessed May 11, 2025, https://github.com/aliakseiherman/mern-multitenancy
24. How I Built a Multi-Tenant Node App - Daniccan Veerapandian, accessed May 11, 2025, https://daniccan.github.io/posts/2020/how-i-built-a-multi-tenant-node-app/
25. Build a Multi-Tenant Architecture in MongoDB - GeeksforGeeks, accessed May 11, 2025, https://www.geeksforgeeks.org/build-a-multi-tenant-architecture-in-mongodb/?ref=oin_asr1
26. How to Implement RBAC in an Express.js Application - Permit.io, accessed May 11, 2025, https://www.permit.io/blog/how-to-implement-rbac-in-an-expressjs-application
27. Implementing Role-Based Access Control (RBAC) in Node.js ..., accessed May 11, 2025, https://www.tenxdeveloper.com/blog/role-based-access-control-in-nodejs-express
28. MERN Stack Security: Best Practices to Keep Hackers at Bay, accessed May 11, 2025, https://www.capitalnumbers.com/blog/mern-stack-security-best-practices/
29. The Rise of JWT Multi Tenant Authentication | Frontegg, accessed May 11, 2025, https://frontegg.com/guides/how-to-persist-jwt-tokens-for-your-saas-application
30. Swagger & Express: Documenting Your Node.js REST API - DEV ..., accessed May 11, 2025, https://dev.to/qbentil/swagger-express-documenting-your-nodejs-rest-api-4lj7
31. Bearer Authentication | Swagger Docs, accessed May 11, 2025, https://swagger.io/docs/specification/v3_0/authentication/bearer-authentication/
32. How to add Swagger OpenAPI docs to NodeJS Express - 93 DAYS, accessed May 11, 2025, https://93days.me/how-to-add-swagger-openapi-docs-to-nodejs-express/
33. Winston Logging in Node.js: The Essential Guide for Developers - Last9, accessed

May 11, 2025, https://last9.io/blog/winston-logging-in-nodejs/
34. A Complete Guide to Winston Logging in Node.js | Better Stack ..., accessed May 11, 2025, https://betterstack.com/community/guides/logging/how-to-install-setup-and-use-winston-and-morgan-to-log-node-js-applications/
35. How to Use Shadcn/UI in React.js - Apidog, accessed May 11, 2025, https://apidog.com/blog/shadcn-ui-in-react-js/
36. Building Accessible UIs with Shadcn UI in React - 4Geeks, accessed May 11, 2025, https://4geeks.com/lesson/building-with-shadcn-ui
37. Shadcn UI React Components | Software.Land, accessed May 11, 2025, https://software.land/shadcn-ui-react-components/
38. Styling and Theming in Shadcn UI - Technical Elements: CSS Properties, Theming APIs, accessed May 11, 2025, https://www.newline.co/@eyalcohen/styling-and-theming-in-shadcn-ui-technical-elements-css-properties-theming-apis--d92efe36
39. Beautiful themes for shadcn/ui — tweakcn | Theme Editor & Generator, accessed May 11, 2025, https://tweakcn.com/
40. Railly/shadcn-ui-customizer - GitHub, accessed May 11, 2025, https://github.com/Railly/shadcn-ui-customizer
41. lucide-react - NPM, accessed May 11, 2025, https://www.npmjs.com/package/lucide-react
42. Lucide Solid, accessed May 11, 2025, https://lucide.dev/guide/packages/lucide-solid
43. Lucide React | Lucide, accessed May 11, 2025, https://lucide.dev/guide/packages/lucide-react
44. Implementing Role Based Access Control (RABC) in React - Permit.io, accessed May 11, 2025, https://www.permit.io/blog/implementing-react-rbac-authorization
45. How to use ReactJS for secure Role Based Access Control | Cerbos, accessed May 11, 2025, https://www.cerbos.dev/blog/how-to-use-react-js-for-secure-role-based-access-control
46. Role-Based Access Control (RBAC) in a MERN Application - DEV Community, accessed May 11, 2025, https://dev.to/nadim_ch0wdhury/role-based-access-control-rbac-in-a-mern-application-5034
47. Best Practices for Multi-Tenant Authorization - Permit.io, accessed May 11, 2025, https://www.permit.io/blog/best-practices-for-multi-tenant-authorization
48. Building a Production Docker Container for Vite + React Apps, accessed May 11, 2025, https://alvincrespo.hashnode.dev/react-vite-production-ready-docker
49. How to Dockerize a React App: A Step-by-Step Guide for Developers, accessed May 11, 2025, https://www.docker.com/blog/how-to-dockerize-react-app/
50. Using npm Workspaces with Docker • Nathan Fries, accessed May 11, 2025, https://nathanfries.com/posts/docker-npm-workspaces/
51. How to containerize a single node app from a mono-repo that utilizes npm workspaces, accessed May 11, 2025,

https://stackoverflow.com/questions/77043969/how-to-containerize-a-single-node-app-from-a-mono-repo-that-utilizes-npm-workspa
52. Step-by-step guide to containerize your full-stack MERN application, using Docker Compose. - DEV Community, accessed May 11, 2025, https://dev.to/uwadon1/step-by-step-guide-to-containerize-your-full-stack-mern-application-using-docker-compose-3a8f
53. Dockerize MERN Application - DEV Community, accessed May 11, 2025, https://dev.to/rakib-dev/dockerize-mern-application-23g5
54. Turning a Node.js Monolith into a Monorepo without Disrupting the Team - InfoQ, accessed May 11, 2025, https://www.infoq.com/articles/nodejs-monorepo/
55. Dockerizing Node.js Apps: A Complete Guide | Better Stack Community, accessed May 11, 2025, https://betterstack.com/community/guides/scaling-nodejs/dockerize-nodejs/
56. Dockerizing a MERN Stack Web Application - GitHub, accessed May 11, 2025, https://github.com/Lucifergene/Docker-Mern
57. Docker Compose in 6 minutes! Mongo, Express, React, Node (MERN) Application Tutorial, accessed May 11, 2025, https://www.youtube.com/watch?v=0B2raYYH2fE
58. Build Containerized MERN App with Lerna Monorepo - DEV Community, accessed May 11, 2025, https://dev.to/rawasthi231/build-containerized-mern-app-with-lerna-monorepo-30d5
59. Master Full-Stack Monorepos: A Step-by-Step Guide - DEV Community, accessed May 11, 2025, https://dev.to/hardikidea/master-full-stack-monorepos-a-step-by-step-guide-2196
60. Dockerizing a MERN Stack Application - Wynn's blog, accessed May 11, 2025, https://wynnt3o.hashnode.dev/dockerizing-a-mern-application-d52ad97e61d3
61. The ABSOLUTE BEST Way to Setup Docker for MERN Stack APP - YouTube, accessed May 11, 2025, https://www.youtube.com/watch?v=LvOz_NfmJzc
62. Dockerizing a MERN App for Development and Production, accessed May 11, 2025, https://zzzachzzz.github.io/blog/dockerizing-a-mern-app-for-development-and-production
63. mern-docker/docker-compose.yml at master · sujaykundu777/mern-docker - GitHub, accessed May 11, 2025, https://github.com/sujaykundu777/mern-docker/blob/master/docker-compose.yml
64. Resources / Best Practices for Using PRDs with Cursor - ChatPRD, accessed May 11, 2025, https://www.chatprd.ai/resources/PRD-for-Cursor
65. Cursor IDE: Setup and Workflow in Larger Projects - Reddit, accessed May 11, 2025, https://www.reddit.com/r/cursor/comments/1ikq9m6/cursor_ide_setup_and_workflow_in_larger_projects/
66. Introducing the Agile-AI Workflow (AIADD): A New Way to Build Complex Apps with Cursor, accessed May 11, 2025,

https://forum.cursor.com/t/introducing-the-agile-ai-workflow-aiadd-a-new-way-to-build-complex-apps-with-cursor/76415
67. Features | Cursor - The AI Code Editor, accessed May 11, 2025, https://www.cursor.com/features
68. Cursor - The AI Code Editor, accessed May 11, 2025, https://www.cursor.com/
69. Cursor AI: Best AI-Powered Coding Assistant For Developers 2025 - Revoyant, accessed May 11, 2025, https://www.revoyant.com/blog/vibe-coding-made-easy-essential-cursor-ai-tool
70. Using the new Project Rules for Monorepos - How To - Cursor - Community Forum, accessed May 11, 2025, https://forum.cursor.com/t/using-the-new-project-rules-for-monorepos/47302
71. Monorepo Project in Cursor - Reddit, accessed May 11, 2025, https://www.reddit.com/r/cursor/comments/1j4p9rs/monorepo_project_in_cursor/
72. Rules - Cursor, accessed May 11, 2025, https://docs.cursor.com/context/rules
73. AISpec + Cursor: Making AI Coding Reliable & Structured (Community RFC + Team Proposal) - Discussion, accessed May 11, 2025, https://forum.cursor.com/t/aispec-cursor-making-ai-coding-reliable-structured-community-rfc-team-proposal/38864
74. How I Built an App with Cursor AI Agent for the First Time (the Good, the Bad, and the Drama) - DEV Community, accessed May 11, 2025, https://dev.to/katya_pavlopoulos/how-i-built-an-app-with-cursor-ai-agent-for-the-first-time-the-good-the-bad-and-the-drama-168o
75. Rules for Shadcn UI - Cursor Directory, accessed May 11, 2025, https://cursor.directory/rules/shadcn-ui
76. The Perfect Cursor AI setup for React and Next.js - Builder.io, accessed May 11, 2025, https://www.builder.io/blog/cursor-ai-tips-react-nextjs
77. How to set up TypeScript with Node.js and Express - LogRocket Blog, accessed May 11, 2025, https://blog.logrocket.com/express-typescript-node/
78. A curated list of awesome .cursorrules files - GitHub, accessed May 11, 2025, https://github.com/PatrickJS/awesome-cursorrules
79. How I use Cursor (+ my best tips) - Builder.io, accessed May 11, 2025, https://www.builder.io/blog/cursor-tips
80. Use Monorepo with backend and frontend or split it? - Cursor - Community Forum, accessed May 11, 2025, https://forum.cursor.com/t/use-monorepo-with-backend-and-frontend-or-split-it/53478
81. CURSOR REVIEW: Debugging with AI Precision - by mason james, accessed May 11, 2025, https://masonjames.com/cursor-review-debugging-with-ai-precision/
82. How to direct Cursor AI to create Docker Compose files for multi-service local development?, accessed May 11, 2025, https://www.rapidevelopers.com/cursor-tutorial/how-to-direct-cursor-ai-to-create-docker-compose-files-for-multi-service-local-development
83. AI That Can Truly Learn and Retain My Codebase - Discussion - Cursor - Community Forum, accessed May 11, 2025, https://forum.cursor.com/t/ai-that-can-truly-learn-and-retain-my-codebase/6740

4
84. dazzaji/Cursor_User_Guide - GitHub, accessed May 11, 2025, https://github.com/dazzaji/Cursor_User_Guide
85. After building +8 PROJECTS with Cursor AI, here's the one trick you really need to know!, accessed May 11, 2025, https://www.reddit.com/r/cursor/comments/1k5uv0f/after_building_8_projects_with_cursor_ai_heres/
86. Awesome Cursor Rules You Can Setup for Your Cursor AI IDE Now - Apidog, accessed May 11, 2025, https://apidog.com/blog/awesome-cursor-rules/
87. Basics of Multi tenant Node.js and PostgreSQL - Clickittech, accessed May 11, 2025, https://www.clickittech.com/saas/multi-tenant-node-js/
88. AdminJS - the leading open-source admin panel for Node.js apps | AdminJS, accessed May 11, 2025, https://adminjs.co/
89. RBAC in multi tenant microservices - Stack Overflow, accessed May 11, 2025, https://stackoverflow.com/questions/62062270/rbac-in-multi-tenant-microservices
90. Building RBAC in Node - Blog - Aserto, accessed May 11, 2025, https://www.aserto.com/blog/building-rbac-in-node
91. Mastering JWT Authentication: A Complete Guide with MERN Stack ..., accessed May 11, 2025, https://dev.to/engrsakib/mastering-jwt-authentication-a-complete-guide-with-mern-stack-4k7g