## RFP

Absolutely, Reza. Here's a professionally rewritten and expanded version of your RFP, incorporating all the clarifications you've provided. This version is structured for clarity, completeness, and future reference during vendor evaluation and development planning.

**🩺 Request for Proposal (RFP)**

**Virtual Patient Training Platform**

**1. Executive Summary**

We are seeking a qualified and experienced development partner to design, build, and deploy a locally hosted, web-based virtual patient training platform. This system will simulate realistic clinical environments and interactions for medical students, enabling immersive, AI-driven training in diagnosis, procedures, and communication. The platform must be scalable, secure, and extensible, with a strong emphasis on realism, modularity, and educational value.

**2. Project Vision & Goals**

Our vision is to revolutionize medical education by providing a high-fidelity, interactive simulation platform that bridges the gap between theory and clinical practice.

**🎯 Primary Goals**

* Deliver a risk-free environment for students to practice clinical decision-making.
* Improve diagnostic accuracy, procedural skills, and patient communication.
* Enable real-time supervision and feedback from educators.
* Support global scalability and institutional customization.

**3. Target Audience**

* **Primary Users**: Medical students, nursing students, and healthcare trainees.
* **Secondary Users**: Clinical supervisors, professors, and administrators.

**4. Core Functional Requirements**

**4.1. User Management**

* Role-based access control (Student, Supervisor, Attendee, Admin).
* Secure authentication via **Keycloak** (SSO-ready).
* Profile management and institutional integration (e.g., SAML).

**4.2. Scenario Engine**

* Manual scenario support for PoC; scalable to dynamic branching logic.
* Patient state modeling: vitals, labs, imaging, nurse reports.
* Support for complications, time acceleration, and attendee overrides.
* Scenario authoring tools for medical specialists (to be developed post-PoC).

**4.3. AI-Powered Interactions**

* Free-text chat with virtual patient and nurse.
* Context-aware responses using locally hosted **LLM via Ollama**.
* NLP pipeline to extract structured orders from student input.
* Emotional cues and patient distress simulation (planned for future phases).

**4.4. 3D Visualization**

* Interactive care room rendered with **Three.js**.
* Clickable patient avatar and instruments.
* Vital signs monitor and ambient sound effects.
* Future support for procedural animations and video-based training.

**4.5. Assessment & Logging**

* Timeline-based logging of all actions (student, nurse, patient, attendee).
* Replay and review capabilities.
* Future scoring engine based on competency dimensions.

**4.6. Real-Time Collaboration**

* Supervisor dashboard for monitoring and intervention.
* Attendee station for adjusting patient state based on student actions.
* WebSocket-based communication between roles.

**4.7. Medical Knowledge Base**

* Searchable database of:
  + Drug information (dosage, interactions).
  + Procedures and protocols.
  + Lab and imaging interpretation guides.

**5. Technical Requirements**

**💻 Technology Stack Preferences**

| **Layer** | **Preferred Technologies** |
| --- | --- |
| Frontend | React or Vue with Three.js |
| Backend | Node.js/NestJS or Python/Django |
| Database | PostgreSQL (Dockerized) |
| AI/LLM | Locally hosted via Ollama (Mistral, Llama 3) |
| Auth | Keycloak |
| Notifications | Novu |
| Real-Time | WebSockets |
| Deployment | Docker Compose for all services |

**🔐 Security**

* JWT-based authentication.
* Data encryption at rest and in transit.
* Protection against OWASP Top 10 vulnerabilities.

**6. Phased Development Plan**

| **Phase** | **Timeline** | **Deliverables** |
| --- | --- | --- |
| **Phase 1: Foundation** | Months 1–3 | Auth system, basic scenario engine, LLM integration, preliminary logging |
| **Phase 2: Core Simulation** | Months 4–6 | Physiology modeling, knowledge base, WebSocket communication, basic 3D room |
| **Phase 3: Enhanced Realism** | Months 7–9 | Advanced AI chat, emotional cues, assessment engine, mobile responsiveness |
| **Phase 4: Production & Polish** | Months 10–12 | Optimization, analytics dashboard, scenario CMS, QA, deployment & documentation |

**7. Vendor Proposal Requirements**

Please include the following in your submission:

1. **Company Overview**: Background, team experience, and relevant expertise.
2. **Portfolio**: Examples of similar projects (AI, 3D, medical, educational).
3. **Technical Proposal**: Architecture, stack justification, and LLM integration strategy.
4. **Project Plan**: Milestones aligned with phased development.
5. **Cost Breakdown**: Detailed pricing per phase, including assumptions.
6. **Team Structure**: Roles and experience of proposed team members.
7. **Project Management**: Methodology (Agile/Scrum), communication cadence.
8. **Support & Maintenance**: Post-launch support, hosting options, and update policy.

**8. Submission Guidelines**

* **Proposal Deadline**: [Insert Date]
* **Submission Email**: [Insert Email]
* **Questions Deadline**: [Insert Date, e.g., 1 week before proposal deadline]
* **Response Format**: PDF or DOCX, max 25 pages

**9. Contact & Communication**

All inquiries must be submitted in writing to the contact above. A consolidated FAQ will be shared with all respondents.

## project creation steps.

I’ll break this down into actionable phases, starting with the proof-of-concept (PoC) and building toward full deployment.

**🧱 Phase 1: Foundation (Months 1–3)**

**1. System Architecture Design**

* Define microservices layout: Auth, Scenario Engine, Chat, AI, 3D, Logging
* Choose communication protocols (REST + WebSocket)
* Design database schema (PostgreSQL): users, scenarios, logs, vitals, labs

**2. Environment Setup**

* Docker Compose for:
  + Keycloak (auth)
  + Novu (notifications)
  + PostgreSQL
  + Ollama (LLM)
  + Backend API
  + Frontend UI

**3. Authentication & Role Management**

* Integrate Keycloak with frontend and backend
* Define roles: Student, Nurse, Attendee, Admin
* Secure endpoints with JWT

**4. Scenario Engine (Basic)**

* Load one manually authored scenario
* Define patient baseline: vitals, history, labs
* Create branching logic template (JSON/YAML)

**5. Chat System**

* Build chat UI for patient and nurse
* Implement NLP pipeline to extract orders from free text
* Route orders to nurse and log actions

**6. LLM Integration**

* Deploy Ollama with chosen model (e.g., Mistral)
* Create prompt templates for patient responses
* Add safety filters for medical appropriateness

**7. Logging & Timeline**

* Create event logger: timestamp, actor, action
* Store logs in PostgreSQL
* Build basic timeline viewer for replay

**🧍 Phase 2: Core Simulation (Months 4–6)**

**1. Physiology Modeling**

* Define rules for medication effects (scripted)
* Update patient vitals based on student actions
* Enable attendee overrides

**2. Medical Knowledge Base**

* Build searchable interface for:
  + Drug info
  + Procedures
  + Lab/imaging guides

**3. WebSocket Communication**

* Enable real-time updates between student and attendee stations
* Allow attendee to inject complications or feedback

**4. 3D Visualization (Basic)**

* Render care room with Three.js
* Add interactive patient avatar
* Display vitals monitor and ambient sounds

**🧠 Phase 3: Enhanced Realism (Months 7–9)**

**1. Advanced AI Chat**

* Add emotional cues to patient responses
* Train LLM on communication best practices

**2. Assessment Engine**

* Define scoring rubric
* Analyze logs for diagnostic accuracy, timing, professionalism
* Generate feedback reports

**3. Mobile Responsiveness**

* Optimize UI for tablets and mobile devices
* Ensure touch interaction in 3D room

**🚀 Phase 4: Production & Polish (Months 10–12)**

**1. Optimization**

* Performance tuning for LLM, 3D rendering, and WebSocket traffic
* Load testing and stress simulation

**2. Analytics Dashboard**

* Visualize student performance trends
* Export reports for supervisors

**3. Scenario CMS**

* Build authoring tool for medical specialists
* Support versioning, tagging, and sharing

**4. QA & Deployment**

* Full test suite: unit, integration, user acceptance
* Final deployment on local server
* Documentation for admins and educators

Let me know if you want me to start drafting the database schema or the system architecture diagram next. Those will be the foundation for development.

## drafting the **system architecture** and **database schema**

drafting the **system architecture** and **database schema** for the proof-of-concept. These will guide the development of your platform’s core modules.

**🧩 System Architecture Overview (PoC)**

**🔄 Architecture Layers**

[Frontend UI] ←→ [Backend API] ←→ [Database]

↓ ↓ ↓

[Three.js 3D Room] [LLM via Ollama] [PostgreSQL]

↓ ↓

[Chat System] [Event Logger]

↓ ↓

[Keycloak Auth] [Novu Notifications]

**🧠 Key Services**

| **Service** | **Description** |
| --- | --- |
| **Frontend** | React/Vue app with chat, 3D room, scenario viewer |
| **Backend** | REST + WebSocket API (Node.js/NestJS or Django) |
| **Auth** | Keycloak (Dockerized) with role-based access |
| **LLM** | Ollama running Mistral or Llama 3 for patient chat |
| **Database** | PostgreSQL for users, scenarios, logs, vitals |
| **Notifications** | Novu for alerts and role-based messages |
| **3D Engine** | Three.js for care room and patient interaction |

**🗃️ Database Schema (PoC)**

**1. Users**

users (

id UUID PRIMARY KEY,

username TEXT,

role ENUM('student', 'nurse', 'attendee', 'admin'),

keycloak\_id TEXT,

created\_at TIMESTAMP

)

**2. Scenarios**

scenarios (

id UUID PRIMARY KEY,

title TEXT,

description TEXT,

initial\_vitals JSONB,

labs JSONB,

imaging JSONB,

nurse\_report TEXT,

branching\_logic JSONB,

created\_by UUID REFERENCES users(id),

created\_at TIMESTAMP

)

**3. Chat Logs**

chat\_logs (

id UUID PRIMARY KEY,

scenario\_id UUID REFERENCES scenarios(id),

sender\_role TEXT,

sender\_id UUID REFERENCES users(id),

message TEXT,

timestamp TIMESTAMP

)

**4. Orders**

orders (

id UUID PRIMARY KEY,

student\_id UUID REFERENCES users(id),

scenario\_id UUID REFERENCES scenarios(id),

order\_type TEXT,

details JSONB,

extracted\_from\_chat BOOLEAN,

timestamp TIMESTAMP

)

**5. Vitals Timeline**

vitals\_log (

id UUID PRIMARY KEY,

scenario\_id UUID REFERENCES scenarios(id),

patient\_state JSONB,

updated\_by UUID REFERENCES users(id),

timestamp TIMESTAMP

)

**6. Event Timeline**

event\_log (

id UUID PRIMARY KEY,

actor\_role TEXT,

actor\_id UUID REFERENCES users(id),

scenario\_id UUID REFERENCES scenarios(id),

event\_type TEXT,

details JSONB,

timestamp TIMESTAMP

)

Next, I can help you define:

* The API endpoints for each module
* The prompt templates for patient AI responses
* The branching logic format for scenarios

Would you like me to continue with the API design or the scenario logic structure next?

## Next.js and NestJS

Perfect choice, Reza. Next.js and NestJS are a powerful combination for building scalable, modern web applications. Here's how we’ll structure your project using these technologies:

**🧱 Updated Tech Stack Overview**

| **Layer** | **Technology** | **Purpose** |
| --- | --- | --- |
| **Frontend** | Next.js | UI rendering, routing, chat interface, 3D visualization |
| **Backend** | NestJS | API services, scenario engine, logging, WebSocket communication |
| **Auth** | Keycloak | Role-based authentication and SSO |
| **AI/LLM** | Ollama | Local LLM for patient interaction |
| **Database** | PostgreSQL | Persistent storage for users, scenarios, logs |
| **Notifications** | Novu | In-app and system alerts |
| **3D Engine** | Three.js | Interactive care room and patient model |
| **Deployment** | Docker Compose | Local orchestration of all services |

**🧩 Project Structure**

**📦 Frontend (Next.js)**

/frontend

/components

ChatPanel.tsx

PatientRoom.tsx

ScenarioMenu.tsx

/pages

/auth

/dashboard

/scenario/[id]

/services

api.ts

websocket.ts

/styles

/utils

**🛠 Backend (NestJS)**

/backend

/src

/auth

/users

/scenarios

/chat

/orders

/vitals

/logs

/llm

/notifications

/websocket

/config

/prisma (if using ORM)

**🔐 Authentication Flow**

* Next.js frontend uses Keycloak JS adapter for login/logout
* NestJS backend validates JWT tokens from Keycloak
* Role-based access enforced at route level

**🔄 Communication**

* REST API for standard data operations
* WebSocket for real-time updates between student and attendee
* LLM service exposed via internal API for patient chat

**🧠 Next Steps**

I can now:

1. Draft the API endpoint structure for NestJS
2. Define the frontend page flow and component responsibilities
3. Create Docker Compose setup for local development

Would you like me to start with the backend API design or the frontend page flow next?

## Script files and folders

Thanks for clarifying, Reza! Here's a simple **shell script** (for Linux/macOS or Windows WSL) that will create the folder and file structure for your **Next.js + NestJS project layout** as discussed:

**🖥️ Shell Script: setup\_project\_structure.sh**

#!/bin/bash

# Create root folders

mkdir -p frontend/components/layout

mkdir -p frontend/pages/scenario

mkdir -p frontend/pages/auth

mkdir -p frontend/styles

mkdir -p frontend/utils

mkdir -p backend/src/{auth,users,scenarios,chat,orders,vitals,logs,llm,notifications,websocket}

mkdir -p backend/config

mkdir -p backend/prisma

# Create frontend layout files

touch frontend/components/layout/{Header.tsx,Sidebar.tsx,Footer.tsx,Layout.tsx}

touch frontend/pages/{index.tsx}

touch frontend/pages/scenario/[id].tsx

touch frontend/pages/auth/{login.tsx,register.tsx}

touch frontend/styles/globals.css

touch frontend/utils/auth.ts

# Create backend module entry files

for module in auth users scenarios chat orders vitals logs llm notifications websocket; do

touch backend/src/$module/{controller.ts,service.ts,module.ts}

done

# Create config and ORM files

touch backend/config/config.ts

touch backend/prisma/schema.prisma

echo "✅ Project structure created successfully!"

**🧾 How to Use**

1. Save the script above as setup\_project\_structure.sh.
2. Open a terminal and navigate to your project root.
3. Run:

chmod +x setup\_project\_structure.sh

./setup\_project\_structure.sh

This will scaffold your folders and placeholder files so you can jump straight into coding.

Would you like me to generate default content for any of these files next (e.g., a basic NestJS module or a Next.js layout component)?

## **bootstrapping the actual codebase**

start **bootstrapping the actual codebase**.

**🚀 Step-by-Step: What’s Next**

**1. Initialize Projects**

#### Frontend (Next.js):

* cd frontend
* npx create-next-app@latest . --typescript
* npm install

#### Backend (NestJS):

* cd ../backend
* npm i -g @nestjs/cli
* nest new . --package-manager npm

**2. Configure Docker Compose**

Set up a docker-compose.yml to run:

* PostgreSQL
* Keycloak
* Novu
* Ollama (LLM)

I can help write this file if you want.

**3. Set Up Keycloak**

* Create a realm (e.g., medical-training)
* Define roles: student, nurse, attendee, admin
* Create a client for your frontend
* Configure JWT validation in NestJS

**4. Build Basic Modules**

**Backend (NestJS)**

* auth: JWT guard + Keycloak integration
* users: Role-based access
* scenarios: Load and serve scenario data
* chat: Accept messages, extract orders
* orders: Store and route structured commands
* logs: Timeline logging

**Frontend (Next.js)**

* Layout components (Header, Sidebar, Footer)
* Pages: Login, Dashboard, Scenario Viewer
* Chat UI: Patient and Nurse panels
* Scenario Menu: Load and switch cases

**5. Connect Frontend to Backend**

* Use Axios or fetch for REST API calls
* Use WebSocket for real-time updates
* Authenticate via Keycloak and pass JWT to backend

**6. Test End-to-End Flow**

* Login as student
* Load a scenario
* Chat with patient and nurse
* Send orders and log actions
* View timeline and vitals

Would you like me to help write the docker-compose.yml next, or scaffold your first NestJS module (e.g., scenarios)?

## linter

For your **Next.js + NestJS** project, I recommend using **ESLint** with a unified configuration that supports both frontend and backend TypeScript code. Here's the best setup for your stack:

#### ✅ Recommended Linter: ESLint with TypeScript Support

**Why ESLint?**

* It's the standard linter for JavaScript/TypeScript projects.
* Fully supported by both **Next.js** and **NestJS**.
* Highly customizable and integrates well with Prettier for formatting.

**🧰 Setup for Each Project**

### 🔹 Frontend (Next.js)

When creating the app, choose **Yes** for ESLint. Then install these additional packages:

npm install --save-dev eslint-config-next eslint-plugin-react eslint-plugin-jsx-a11y eslint-plugin-import

Your .eslintrc.json might look like:

{

"extends": ["next/core-web-vitals", "plugin:react/recommended", "plugin:jsx-a11y/recommended"],

"rules": {

"react/react-in-jsx-scope": "off"

}

}

### 🔸 Backend (NestJS)

##### Install ESLint and TypeScript support:

npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin

Then create .eslintrc.js:

module.exports = {

parser: '@typescript-eslint/parser',

parserOptions: {

project: 'tsconfig.json',

sourceType: 'module',

},

plugins: ['@typescript-eslint'],

extends: [

'plugin:@typescript-eslint/recommended',

'plugin:@typescript-eslint/recommended-requiring-type-checking',

],

root: true,

env: {

node: true,

jest: true,

},

rules: {

'@typescript-eslint/no-unused-vars': ['warn'],

'@typescript-eslint/explicit-function-return-type': 'off',

},

};

### ✨ Optional: Add Prettier for Formatting

Install Prettier and integrate it with ESLint:

npm install --save-dev prettier eslint-config-prettier eslint-plugin-prettier

Then extend your ESLint config:

"extends": [

"plugin:prettier/recommended"

]

Would you like me to generate a shared .eslintrc and .prettierrc file for both projects to keep them consistent?

For your project, Reza, I recommend selecting **No** when asked about using the React Compiler during Next.js setup — at least for now.

#### 🧠 Why Skip React Compiler (for now)?

* **React Compiler is experimental**: It’s a new optimization tool that rewrites React components for performance, but it's not yet stable or widely supported.
* **You’re using TypeScript + Three.js + LLM integration**: These are already complex and performance-sensitive. Adding experimental tooling could introduce unexpected issues.
* **Better control with manual optimization**: You can fine-tune performance later using memoization, lazy loading, and server-side rendering.

**✅ Recommended Answer**

**Would you like to use React Compiler?** → No

Once your platform is stable and you're ready to optimize rendering performance, we can revisit React Compiler or alternatives like React Server Components or fine-grained memoization.

Would you like help configuring Tailwind CSS or setting up your first page layout next?

For your project, Reza, I recommend selecting **No** when asked:

#### Would you like to use the App Router?

**🧠 Why Choose “No” (Use Pages Router Instead)**

* You're building a **role-based, multi-user platform** with structured routing (e.g., /auth/login, /scenario/[id], /dashboard). The **Pages Router** is simpler and more predictable for this.
* The **App Router** is part of Next.js’s newer architecture (based on React Server Components), but it’s still evolving and may introduce complexity with:
  + Authentication flows (especially with Keycloak)
  + WebSocket integration
  + Dynamic routing for scenarios and roles

**✅ Recommended Answer**

**Would you like to use App Router?** → No

You’ll still get full support for SSR, API routes, and dynamic pages — all of which are ideal for your platform.

Let me know when you're ready to scaffold your first page or connect Keycloak to your frontend.

#### For your project, Reza, I recommend selecting **No** when asked:

**Would you like to use Turbopack?**

**🧠 Why Choose “No” (Stick with Webpack for Now)**

* **Turbopack is experimental**: It’s a new Rust-based bundler aiming to replace Webpack, but it’s still under active development and may not support all features or plugins you’ll need.
* **Your stack is complex**: You’re integrating Three.js, Keycloak, WebSockets, and LLMs — sticking with the stable Webpack ensures compatibility and easier debugging.
* **Better community support**: Most Next.js plugins, tutorials, and tools are still built around Webpack.

**✅ Recommended Answer**

**Would you like to use Turbopack?** → No

You’ll still get fast builds and full compatibility with your chosen tech stack. Later, if Turbopack becomes stable and widely supported, you can explore switching.

Let me know when you're ready to start building your first page or connect the frontend to Keycloak.

## SharedTypes project

Excellent questions, Reza — you're thinking like a true architect. Let's break this down into two key areas: **database setup** and **shared types strategy**.

**🗃️ 1. Database Setup: When and How**

**✅ When to Create the Database**

You should create the database **now**, during the early backend setup phase. This allows you to:

* Define your schema
* Scaffold your NestJS services and DTOs
* Seed initial data for testing scenarios

**🧰 How to Set It Up**

Use **PostgreSQL** via Docker Compose for local development:

##### # docker-compose.yml

services:

postgres:

image: postgres:15

restart: always

environment:

POSTGRES\_USER: medsim

POSTGRES\_PASSWORD: securepass

POSTGRES\_DB: medsim\_db

ports:

- "5432:5432"

volumes:

- pgdata:/var/lib/postgresql/data

volumes:

pgdata:

Then connect from NestJS using @nestjs/typeorm or @nestjs/sequelize, or use Prisma if you prefer a schema-first approach.

#### 🔁 2. Shared Types Between Backend and Frontend

**✅ Should You Create a sharedTypes Project?**

**Yes — absolutely.** This is a best practice for projects like yours where:

* You have structured data (e.g., Scenario, Vitals, Order, ChatMessage)
* You want type safety across API boundaries
* You’re using TypeScript on both ends

**🧱 How to Structure It**

##### Create a third workspace folder:

/sharedTypes

/src

scenario.ts

vitals.ts

order.ts

chat.ts

tsconfig.json

package.json

Then:

* Use tsup or tsc to compile it
* Add it as a dependency to both frontend and backend
* Use npm link or a monorepo tool like **pnpm workspaces**, **Yarn workspaces**, or **Nx** for local development

#### 🧠 Bonus: Use Zod or io-ts

If you want runtime validation and shared schemas, consider using zod:

##### // sharedTypes/src/scenario.ts

import { z } from 'zod';

export const ScenarioSchema = z.object({

id: z.string().uuid(),

title: z.string(),

initialVitals: z.record(z.string(), z.number()),

labs: z.array(z.string()),

});

export type Scenario = z.infer<typeof ScenarioSchema>;

This gives you:

* Type safety
* Runtime validation
* Shared usage in NestJS DTOs and Next.js forms

Would you like me to scaffold the sharedTypes project structure or help define your first shared model (e.g., Scenario)?

## **database setup and seeding**

Perfect, Reza — let’s build this out step by step. We’ll start with the **sharedTypes** project, then move into **database setup and seeding**, and finally continue with backend scaffolding.

**🧩 Step 1: Create sharedTypes Project**

#### 📁 Folder Structure

Place this alongside your frontend and backend folders:

/sharedTypes

/src

scenario.ts

vitals.ts

order.ts

chat.ts

tsconfig.json

package.json

**📦 Initialize the Project**

mkdir sharedTypes

cd sharedTypes

npm init -y

npm install typescript --save-dev

npx tsc --init

#### 🛠 Update tsconfig.json

{

"compilerOptions": {

"target": "ES2020",

"module": "CommonJS",

"declaration": true,

"outDir": "./dist",

"rootDir": "./src",

"strict": true,

"esModuleInterop": true

},

"include": ["src"]

}

#### 🧪 Sample Type: scenario.ts

// sharedTypes/src/scenario.ts

export interface Scenario {

id: string;

title: string;

description: string;

initialVitals: Record<string, number>;

labs: string[];

imaging: string[];

nurseReport: string;

branchingLogic: Record<string, any>;

}

#### 🔗 Link to Frontend and Backend

In both frontend and backend:

npm install --save ../sharedTypes

Or use a monorepo tool like pnpm or Yarn workspaces for better dev experience.

**🗃️ Step 2: Initialize PostgreSQL Database**

##### 🐘 Add to docker-compose.yml

postgres:

image: postgres:15

restart: always

environment:

POSTGRES\_USER: medsim

POSTGRES\_PASSWORD: securepass

POSTGRES\_DB: medsim\_db

ports:

- "5432:5432"

volumes:

- pgdata:/var/lib/postgresql/data

volumes:

pgdata:

#### 🧱 Connect from NestJS

Install TypeORM or Prisma (I’ll use TypeORM here):

npm install @nestjs/typeorm typeorm pg

Update AppModule:

TypeOrmModule.forRoot({

type: 'postgres',

host: 'localhost',

port: 5432,

username: 'medsim',

password: 'securepass',

database: 'medsim\_db',

autoLoadEntities: true,

synchronize: true,

});

**🌱 Step 3: Seed Initial Data**

**🧪 Create Entity: Scenario**

// backend/src/scenarios/scenario.entity.ts

import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()

export class Scenario {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

title: string;

@Column()

description: string;

@Column('jsonb')

initialVitals: Record<string, number>;

@Column('jsonb')

labs: string[];

@Column('jsonb')

imaging: string[];

@Column()

nurseReport: string;

@Column('jsonb')

branchingLogic: Record<string, any>;

}

**🌱 Create Seeder Script**

// backend/src/scenarios/scenario.seed.ts

import { DataSource } from 'typeorm';

import { Scenario } from './scenario.entity';

const dataSource = new DataSource({

type: 'postgres',

host: 'localhost',

port: 5432,

username: 'medsim',

password: 'securepass',

database: 'medsim\_db',

entities: [Scenario],

synchronize: true,

});

dataSource.initialize().then(async () => {

const repo = dataSource.getRepository(Scenario);

await repo.save({

title: 'Chest Pain Evaluation',

description: 'Patient presents with acute chest pain...',

initialVitals: { heartRate: 110, bp: 140 },

labs: ['Troponin', 'CBC'],

imaging: ['Chest X-ray'],

nurseReport: 'Patient is anxious and sweating.',

branchingLogic: {},

});

console.log('✅ Seeded scenario');

process.exit();

});

Run it with:

ts-node src/scenarios/scenario.seed.ts

**🧭 Next Steps**

Now that you have:

* Shared types
* Database running
* Seeded data

You can:

1. Scaffold the ScenarioService and ScenarioController in NestJS
2. Build the frontend page to fetch and display scenarios
3. Add Keycloak authentication

Would you like me to scaffold the NestJS ScenarioModule next or help build the frontend scenario viewer?

## **TypeORM**

Perfect — let’s continue building your backend using **TypeORM** in NestJS. You’ve already initialized the project and seeded the database. Now we’ll:

### 🧱 Step 4: Scaffold the ScenarioModule in NestJS

This module will:

* Expose REST endpoints to fetch scenarios
* Use the Scenario entity (already defined)
* Be ready for future features like scenario creation and updates

### 📁 Folder Structure

/backend/src/scenarios

scenario.entity.ts

scenario.service.ts

scenario.controller.ts

scenario.module.ts

#### 🧾 scenario.module.ts

import { Module } from '@nestjs/common';

import { TypeOrmModule } from '@nestjs/typeorm';

import { Scenario } from './scenario.entity';

import { ScenarioService } from './scenario.service';

import { ScenarioController } from './scenario.controller';

@Module({

imports: [TypeOrmModule.forFeature([Scenario])],

providers: [ScenarioService],

controllers: [ScenarioController],

})

export class ScenarioModule {}

#### 🧾 scenario.service.ts

import { Injectable } from '@nestjs/common';

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { Scenario } from './scenario.entity';

@Injectable()

export class ScenarioService {

constructor(

@InjectRepository(Scenario)

private readonly scenarioRepo: Repository<Scenario>,

) {}

findAll(): Promise<Scenario[]> {

return this.scenarioRepo.find();

}

findOne(id: string): Promise<Scenario | null> {

return this.scenarioRepo.findOneBy({ id });

}

}

#### 🧾 scenario.controller.ts

import { Controller, Get, Param } from '@nestjs/common';

import { ScenarioService } from './scenario.service';

@Controller('scenarios')

export class ScenarioController {

constructor(private readonly scenarioService: ScenarioService) {}

@Get()

getAll() {

return this.scenarioService.findAll();

}

@Get(':id')

getOne(@Param('id') id: string) {

return this.scenarioService.findOne(id);

}

}

#### 🧩 Register in AppModule

import { ScenarioModule } from './scenarios/scenario.module';

@Module({

imports: [

TypeOrmModule.forRoot({...}),

ScenarioModule,

],

})

export class AppModule {}

#### ✅ Test It

Start your backend:

npm run start:dev

Then test:

* GET http://localhost:3000/scenarios → list all scenarios
* GET http://localhost:3000/scenarios/<id> → get one scenario

**🧭 Next Steps**

Now that your backend is serving scenario data, we can:

1. Build the frontend page to fetch and display scenarios
2. Add Keycloak authentication to protect routes
3. Add chat and order modules next

Would you like to build the frontend scenario viewer next, or scaffold another backend module like chat or orders?

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1

## S1