A smart and practical extension of existing tools like AutoCAD Plant 3D integrated with SQL Server. Let’s break down **why** they’re building a **full stack project**, what the **purpose** is, and what the **advantages** are.

**🧠 Purpose of the Full Stack Project**

Even though AutoCAD Plant 3D is connected directly to SQL Server, it's mainly a **design tool** focused on modeling and drafting. It's not built for **data management, analysis, inventory tracking, or business workflows**. Here’s what your friend's full stack system likely aims to do:

1. **Centralized Management of Parts Data**
   * Allow engineers, procurement, warehouse, and QA teams to access part details **outside AutoCAD** in a web-friendly interface.
2. **Inventory & Lifecycle Tracking**
   * Track quantities, statuses, stock levels, usage history, maintenance logs, etc., for each machinery part.
3. **Enhanced CRUD Operations**
   * Provide a user-friendly UI to **add, edit, or delete parts**, update fluid specs, or link them to specific projects or locations.
4. **Cross-Department Collaboration**
   * Make it easy for non-technical staff (managers, inventory controllers, inspectors) to view or update data without needing AutoCAD skills.
5. **Export and Reporting**
   * Generate **PDF reports or Excel sheets** for engineering reviews, audits, procurement, or inspections.
6. **Custom Workflows**
   * Implement company-specific rules like approval processes, status flags (e.g., "Under Review", "Approved", "Obsolete"), or version control for part data.
7. **Scalable Access**
   * Host the system on a web server so it can be accessed by authorized staff from **multiple departments or even remote locations**.

**✅ Advantages of Building This Full Stack System**

| **Feature** | **Advantage** |
| --- | --- |
| 🌐 **Web Interface** | Access data from any browser without needing AutoCAD installed. |
| 👩‍💼 **Role-Based Access** | Engineers can edit, managers can approve, warehouse staff can view inventory—each with different access levels. |
| 📊 **Data Analytics** | Track usage frequency, downtime trends, or fluid compatibility issues for certain parts. |
| 📤 **Export Options** | Convert part records into PDF or Excel with one click for reporting, documentation, or compliance. |
| 🧩 **Extendability** | Can be integrated with other systems like ERP, maintenance systems, or purchase orders. |
| 🚫 **Reduced AutoCAD Dependence** | AutoCAD is heavy software—your system allows users to work with the data without opening drawings. |
| 🔒 **Audit & Version Control** | Log who changed what and when. Useful for quality control and ISO certification processes. |

**🎯 Real-World Example**

Let’s say an engineer adds a **rotary pump** in AutoCAD Plant 3D, and it's saved in the SQL Server. Later on:

* A **warehouse staff** logs into the web app and sees there are 3 such pumps in stock.
* A **procurement officer** downloads a list of pumps needing replacement next quarter.
* A **QA officer** flags one of the pumps as "corrosive failure risk" based on fluid analysis.
* The **engineering manager** generates a PDF report summarizing all rotary pump specs for client documentation.

All of this is possible **without touching AutoCAD**—that’s the power of a full stack project built around the core design tool.

**🧠 Summary**

Even though AutoCAD + SQL Server captures the raw data, a full stack system:

* **Makes the data usable across departments**
* **Adds business logic and workflows**
* **Extends reporting, exporting, and usability**
* **Increases collaboration and accessibility**

**🚀 1. Ways to Further Improve the Full Stack System with AI**

Here are **AI-powered features** your friend could add:

**🛠️ Predictive Maintenance**

* Use historical data (like usage hours, fluid type, operating temperature) to **predict when a pump or tank might fail or need servicing**.
* Train a machine learning model on past failure records.

🔧 *"This rotary pump has an 82% likelihood of failure within the next 2 months based on historical trends."*

**📈 Smart Inventory Optimization**

* AI can analyze usage patterns and suggest **optimal inventory levels**, helping prevent overstocking or understocking.

🧮 *"You should order 2 vacuum pumps within 3 weeks based on projected usage and lead time."*

**🔍 Natural Language Search**

* Allow engineers to **search parts using natural language** instead of filters.

Example: *"Show me all pumps that can handle corrosive fluids above 100°C with density over 900 kg/m³."*

**📎 AI-Powered Data Entry Assistance**

* When adding a new part, AI can **auto-fill common fields** (like standard viscosity ranges or default fluids) based on similar records.

✨ *"It looks like this is a centrifugal pump. Do you want to auto-fill common attributes?"*

**🔐 Anomaly Detection**

* Detect data issues or unusual patterns, such as:
  + A part that was used in the wrong environment
  + A sudden spike in a fluid's viscosity that doesn’t match norms

⚠️ *"This entry shows an unusually high density for water. Are you sure it's correct?"*

**📊 AI Summary Reports**

* Generate human-readable summaries of technical data for supervisors or non-engineers.

🧾 *"There are 13 active agitators operating within spec, and 2 out-of-spec due to fluid temperature exceeding 130°C."*

**🧠 Chatbot or Assistant for Engineers**

* Add a sidebar chatbot that can:
  + Answer questions like *"What parts were used in the March 2023 expansion project?"*
  + Help navigate the system quickly

**🌟 2. How AI Can *Improve* (Not Just Add Features to) the System**

Sometimes it’s not about adding a feature, but **making existing ones smarter**:

| **Current Feature** | **How AI Can Improve It** |
| --- | --- |
| **PDF/Excel Reports** | Auto-generate summaries, highlight critical issues, or flag missing fields |
| **Data Validation** | Use AI to flag improbable values (e.g., temp too high for known materials) |
| **User Experience (UX)** | Personalized dashboards based on roles or recent activity |
| **Search Filters** | Intelligent filtering based on previous user behavior or common queries |
| **Form Inputs** | AI can suggest dropdown defaults or warn about conflicting inputs |
| **Approval Flows** | Prioritize parts needing attention based on risk or usage score |

**🧩 Realistic Implementation Stack (Example)**

| **Area** | **Tech** |
| --- | --- |
| Machine Learning | Python (scikit-learn), TensorFlow, or PyTorch for training models |
| Prediction API | Flask or FastAPI to expose models to your full stack system |
| AI Search | ElasticSearch + NLP models (like OpenAI embeddings or Hugging Face transformers) |
| Chatbot | OpenAI API, LangChain, or Rasa integrated into React frontend |
| Anomaly Detection | Isolation Forest, Autoencoders, or One-Class SVM models trained on system data |

**🔚 Summary**

**Yes**, AI can absolutely enhance the system in both **new features** and **smarter behavior**:

* Improve accuracy and efficiency
* Help predict issues and reduce downtime
* Make the system more intuitive and user-friendly
* Support better decision-making with intelligent summaries

Next Steps:  
✅ Review the structure for best practices

1. 🔧 Suggest improvements (e.g., normalization, naming, indexing)
2. ➕ Recommend additional tables to support features like:
   * Inventory tracking
   * Part specifications (temp, viscosity, etc.)
   * Maintenance history
   * Export logging
   * AI-related data collection

Once the schema is clean and scalable, we can move to:

* Backend setup using **Next.js API routes or Express**
* Implementing **basic features** (CRUD, search, inventory)
* Adding **PDF/Excel export**
* Then preparing the structure for **AI features** down the road

**1. Where should change logs appear?**

You have two types of change tracking:

**🔹 A. Inline edits of InformationValues**

These changes (value + UOM) go into InformationChangeLogs.

**🔹 B. View logs per datasheet revision?**

Yes! You can show them **inside each datasheet view**, maybe via:

* A **“Change History”** collapsible section
* A **tab** next to "Equipment Details"
* Or an **Audit Trail modal** with:

ts

CopyEdit

🔄 ChangedBy: "engineer@example.com"

🕓 ChangedAt: April 21, 2025 – 10:37 AM

🧩 Template: "Operating Pressure"

🔁 Value: 101 psi → 698 kPa

That gives users full context while viewing the revision.

**🦅 2. Bird’s Eye View of Datasheets & Revisions**

Let’s design this smartly.

**🧩 Assume your Sheets table looks like:**

ts

CopyEdit

SheetID | ParentSheetID | RevisionNum | Status | IsLatest | SheetNameEng

--------|----------------|-------------|-----------|----------|--------------

3 | NULL | 0 | Approved | ❌ | Pump-001

4 | 3 | 1 | Draft | ✅ | Pump-001

**🖼 What You Can Build: “Datasheet Revision Browser”**

**✅ View All Parent Datasheets:**

sql

CopyEdit

SELECT \* FROM Sheets WHERE ParentSheetID IS NULL;

Each row is a **base datasheet**.

**🧭 Show Revisions with:**

sql

CopyEdit

SELECT \* FROM Sheets WHERE ParentSheetID = @SheetID ORDER BY RevisionNum DESC;

Now display something like this:

scss

CopyEdit

📄 Pump-001

🔢 Rev 1 (Draft) 🟡 Editable ✅ IsLatest

🔢 Rev 0 (Approved) 🟢 Read-only

Add buttons:

* 🔍 View
* 📄 Duplicate into New Draft
* 🕵️ View Audit Log

You can even use collapsible cards or timelines to visualize revisions.

**🪄 Bonus: Grouping View by Status**

A dashboard or toggle view to show:

* ✅ **Approved Datasheets**
* 🛠 **In Progress** (Drafts, Verified)
* 🧭 **Datasheets with pending revisions**

You already have everything — you just need to surface it smartly in UI.

**✅ What We Can Do Next**

Would you like help building:

1. A **/datasheets list page** with expandable revisions?
2. A **“Change History” modal** on each datasheet revision?
3. A **backend route** to fetch all InformationChangeLogs for a sheet?

Your system is shaping up to be **enterprise-grade**, with:

* Full audit trail visibility 🕵️‍♂️
* Intelligent revision tracking 📄
* Clean user/role control 🔒
* Multilingual exports 🌐
* Future AI features on the roadmap 🤖

Let’s keep going at your pace. Ready to move forward with:

* Inline editing logic
* Saving changes with change logs
* Or building the “Revision Browser” view?

Just point the way 👇

**🧱 Overview of Datasheet Types**

| **Type** | **Description** | **Visibility** | **Editable By** |
| --- | --- | --- | --- |
| 📄 **Templates** | Unfilled master datasheets (structure only) | All users (view), Admin (edit) | Admin/Supervisor |
| 📘 **Filled Datasheets** | Instances with actual values from InformationValues | All users | Drafts: Assigned user Approved: Read-only |
| 📂 **Revisions** | Historical or in-progress versions of filled datasheets (versioned copies) | All users (grouped per parent) | Depends on Status |

**🌐 Recommended Page Layouts & Routes**

**1. ✅ /datasheets/templates**

* **Description**: List of all **unfilled templates**
* **Features**:
  + View: 🧐 Template details (readonly)
  + Admin-only:
    - ✏️ Edit Template
    - ➕ Create Template
    - 📤 Export (PDF/Excel)
* **Route Examples**:
  + /datasheets/templates — list view
  + /datasheets/templates/45 — detail view
  + /datasheets/templates/45/edit — editable form
  + /datasheets/templates/create — create new

**2. 📘 /datasheets/filled**

* **Description**: Actual datasheets filled by engineers or AutoCAD sync
* **Features**:
  + 🔍 Filter by Client, Category, Status
  + 📝 Inline edit (if Draft)
  + 📤 Export to PDF/Excel
  + ⏪ View Revision History
* **Route Examples**:
  + /datasheets/filled — list view
  + /datasheets/filled/78 — view/edit single filled datasheet
  + /datasheets/filled/78/logs — change history
  + /datasheets/filled/78/revisions — revision browser

**3. 🕵️‍♂️ /datasheets/revisions**

* **Description**: Bird’s-eye view of **all datasheet revisions**
* **Group by**: ParentSheetID
* **Show**:
  + Parent name
  + Each revision (status, version, isLatest)
  + Actions:
    - 👁 View
    - 🧬 Duplicate
    - 🕵️ Logs

**🧩 Suggested Sidebar Menu Structure**

bash

CopyEdit

📂 Datasheets

├── 🧩 Templates → /datasheets/templates

│ └── ➕ Create Template → /datasheets/templates/create

├── 📘 Filled Forms → /datasheets/filled

├── 🧬 Revisions → /datasheets/revisions

**🧠 UX Enhancements by User Role**

| **Role** | **Views Enabled** | **Actions Available** |
| --- | --- | --- |
| Admin | All pages | Full edit, versioning, create templates |
| Engineer | Filled + Revisions | Edit drafts, duplicate, view logs |
| Manager | Filled + Revisions | Approve, review |
| Viewer | Templates (view), Filled (readonly) | None |

**🔍 Filtering & Status Toggles**

Include a **toggle bar or filter section** on list views:

* 🔘 **Status**: Draft / Verified / Approved
* 🔘 **Project / Category / Client**
* 🔘 **With Pending Revision**

**📊 Future Dashboard Ideas**

* ⏱ **Recently Edited Datasheets**
* 📌 **Pending Approvals**
* ⚠️ **Stale Revisions / Unused Templates**
* 📈 **Top Used Templates / Categories**

**✅ Immediate Next Steps**

1. **Split your routes** into:
   * /datasheets/templates
   * /datasheets/filled
   * /datasheets/revisions
2. **Add filters** (status, client, category) to each list view.
3. **Build revision browser** with collapsible UI per parent.
4. Add permissions:
   * **Admin-only**: create/edit templates
   * **Draft-only**: enable inline editing

**🌐 Suggested Sidebar Menu Structure**

pgsql

CopyEdit

📁 Dashboard

├── 📊 Overview & Stats

├── 🔔 Alerts & Warnings (AI)

├── 🛠️ Maintenance Summary (Future AI)

📂 Datasheets

├── 🧩 Templates (Unfilled)

│ ├── ➕ Create Template

│ └── 📋 View All Templates

├── 📘 Filled Datasheets

│ └── 📥 Upload / Fill New Sheet

├── 🧬 Revisions

│ └── 📜 View All Versions

├── 🕵️ Audit Logs

│ └── ⏱️ Change History

📦 Inventory

├── 📇 Parts & Components

├── 📍 Stock Levels

├── 🛒 Procurement Tracking

├── 📆 Maintenance Logs (Future)

├── 🧠 AI Predictions (Future)

📑 Reports & Exports

├── 📤 Export to PDF / Excel

├── 📈 Analytics & KPIs

├── 📎 Generate Summaries (AI)

👥 System Access

├── 👤 Users & Roles

├── 🔒 Permissions Matrix

⚙️ Settings

├── 🌍 Language & Units

├── 🔄 Integration Settings (ERP, AutoCAD)

├── 🧠 AI Configurations (Future)

🧠 AI Assistant

└── 🤖 Chat with SpecBot (Contextual Q&A, Search)

**🔍 Key Highlights & Justifications**

| **Section** | **Why It’s Useful** |
| --- | --- |
| **Dashboard** | Immediate overview for managers, engineers, QA with KPIs & AI insights. |
| **Templates** | Admins manage the unfilled structure. Clearly separated to prevent misuse. |
| **Filled Datasheets** | Engineers view and fill; managers approve; works across workflows. |
| **Revisions** | Tracks all versions — clean revision history interface for quality control. |
| **Inventory** | Warehouse, procurement, engineering use this daily; integrates well with AI. |
| **Reports & Exports** | One-click PDF/Excel for audits, clients, and documentation. |
| **System Access** | Scales well for RBAC: show/hide menus by role. |
| **Settings** | Central place for multi-language, units, and integrations. |
| **AI Assistant** | Bonus AI sidebar tool — natural language search, prediction summaries, etc. |

**✅ Role-Based Sidebar Display Example**

| **Role** | **Visible Menus** |
| --- | --- |
| **Admin** | All menus |
| **Engineer** | Dashboard, Filled Datasheets, Revisions, Inventory, Reports |
| **QA** | Dashboard, Filled Datasheets, Audit Logs, Reports |
| **Manager** | Dashboard, Revisions, Reports, Approvals |
| **Viewer** | Dashboard, Filled Datasheets (read-only), Reports |

**🧠 Future-Proofing for AI**

With your vision for:

* AI-powered predictive maintenance
* Smart inventory alerts
* NLP-based search
* Anomaly detection

The **AI Assistant** and **Maintenance Logs** menu items prepare your system to gradually evolve into an intelligent tool, not just a record-keeping system.

**💡 Suggested Presentation and Features**

**🔹 /estimation/dashboard**

* Visual summary of recent estimates, grouped by:
  + Project
  + Client
  + Status (Pending, Finalized, Sent)
* KPIs: Total Estimated Cost, Materials vs Labor split, Estimation Accuracy

**🔹 /estimation/create**

A guided form that lets you:

* Select a project or datasheet
* Pull in equipment lists and quantities automatically from datasheets
* Add cost per unit, labor rates, contingency %
* Auto-calculate:
  + Total Material Cost
  + Labor Cost
  + Final Estimate (with markup/contingency)
* Save or export as PDF/Excel

**🔹 /estimation/history**

* List of all saved estimates
* Filter by client, project, date
* View/Download button
* "Duplicate" feature to reuse similar estimates

**🧠 Future Enhancements with AI**

Later on, you can add:

* **AI-powered suggestions** for unit prices or labor cost based on past data
* **Auto-generated estimation reports** in plain language
* **Anomaly detection**: "This estimate seems unusually low for this category."

**✅ Summary of SpecVerse with react-i18next**

| **Your current** | **With i18next** |
| --- | --- |
| UILabelTranslations table | ✅ Remove |
| SheetTranslations table | ✅ Remove |
| getLabel() function | ✅ Replace with t() |
| languages table | ✅ Replace with i18n.changeLanguage() |
| Translations stored in DB | ✅ Now file-based (/src/locales/) |
| Changing language = reload | ✅ Changing language = instant |
| Complex API queries | ✅ No API queries |

👉 Result: Cleaner, lighter, and easier to maintain + fully compatible with Next.js SSR or CSR.

**✅ Real-world companies using i18next**

react-i18next is used by:

* Microsoft
* SAP
* Zalando
* Allianz
* Many major enterprise Next.js apps

**✅ 🎯 PHASE 1 — Prepare i18next setup**

1. Install react-i18next and dependencies:

bash

CopyEdit

npm install react-i18next i18next i18next-browser-languagedetector

1. Create translation files:

swift

CopyEdit

/src/locales/en/common.json

/src/locales/fr/common.json

... (other languages as you add them)

1. Add your current database labels to these files gradually.
2. Create src/i18n.ts and initialize i18next:

* This file holds config + loads translations from /src/locales/.

1. Wrap your app in I18nextProvider in src/app/layout.tsx (or \_app.tsx if older Next.js).

👉 Your DB translation system keeps working **as backup** during this phase.

**✅ 🎯 PHASE 2 — Create unified t() wrapper**

To transition safely:

1. Replace your current getLabel() with a custom hook:

tsx

CopyEdit

import { useTranslation } from "react-i18next";

export function useUILanguage() {

const { t } = useTranslation();

return (key: string) => t(key);

}

1. Update your UI:

tsx

CopyEdit

const getLabel = useUILanguage();

<h2>{getLabel("Datasheet Details")}</h2>

✅ This allows you to switch from DB to i18next gradually.

**✅ 🎯 PHASE 3 — Slowly migrate labels**

1. Begin replacing most common UILabelTranslations lookups with t().
2. Migrate all subsheet labels and template labels into JSON files.
3. Keep database fallback active for older records if needed (?? databaseLabel).

✅ No downtime. Users won’t even notice you’re migrating.

**✅ 🎯 PHASE 4 — Migrate language switcher**

1. Replace your current database-driven language select:

tsx

CopyEdit

i18n.changeLanguage("fr");

1. Remove the need for /api/languages.

✅ Changing language becomes instant. No backend query needed.

**✅ 🎯 PHASE 5 — Clean up database + APIs**

Once all translations are stable:

1. Drop or archive:
   * UILabelTranslations
   * SheetTranslations
   * SubsheetTranslations
   * Any translations related queries
2. Remove any backend routes related to GET /ui-labels or GET /subsheets/translated.

✅ Drastically reduces your DB size + API complexity.  
✅ Maintenance is now only file-based inside Git.

**✅ 🎯 PHASE 6 — Add developer-friendly features**

1. Create a reusable useLanguage() hook for:
   * get current language
   * change language
2. Document your translation rules (recommended key structure, file locations).
3. Optionally add dynamic namespace support (t('datasheets:Save') etc.)

**✅ FINAL BENEFITS for SpecVerse**

| **Before** | **After** |
| --- | --- |
| Labels from DB | Labels from files |
| Slow language switching | Instant switching |
| Requires SQL queries | Pure front-end |
| Complex API routes | Clean React hooks |
| Harder to manage versions | Easy to version control with Git |
| Translation updates require DB access | Anyone with Git can update translations |

This is **exactly how enterprise apps like SAP, Microsoft, Atlassian do it.**

You will have:

* 🚀 **less code**
* 🚀 **smaller DB**
* 🚀 **faster UI**
* 🚀 **simpler developer experience**

**🎉 Summary of recommended migration order**

| **Step** | **Task** |
| --- | --- |
| 1 | Install + set up i18next |
| 2 | Replace getLabel() with useUILanguage() hook |
| 3 | Slowly migrate translation keys to files |
| 4 | Switch language selector to i18n.changeLanguage() |
| 5 | Remove DB + backend APIs for translations |
| 6 | (Optional) Add dynamic namespaces + dev documentation |

This plan is safe, iterative, with **zero risk of downtime.**

SpecVerse/

├── public/

│ └── clients/ → client logos/images

├── src/

│ ├── app/

│ │ ├── (admin)/

│ │ │ ├── datasheets/

│ │ │ │ ├── filled/

│ │ │ │ │ └── [id]/page.tsx

│ │ │ │ ├── templates/

│ │ │ │ │ ├── create/page.tsx

│ │ │ │ │ └── [id]/page.tsx

│ │ │ ├── clients/

│ │ │ ├── projects/

│ │ │ ├── categories/

│ │ │ ├── users/

│ │ │ └── reports/

│ │ ├── layout.tsx → includes <I18nextProvider />

│ │ └── globals.css

│ ├── components/

│ │ ├── datasheets/

│ │ │ ├── templates/

│ │ │ │ ├── TemplateCreatorForm.tsx

│ │ │ │ ├── SubsheetBuilder.tsx

│ │ │ │ ├── InfoTemplateBuilder.tsx

│ │ ├── common/

│ │ │ ├── LanguageSwitcher.tsx ✅ new reusable component

│ │ │ ├── AppHeader.tsx

│ │ │ ├── AppSidebar.tsx

│ ├── hooks/

│ │ ├── useUILanguage.ts ✅ your wrapper for t()

│ │ ├── useUserRole.ts

│ ├── locales/ ✅ NEW - i18next translations

│ │ ├── en/

│ │ │ └── common.json

│ │ ├── fr/

│ │ │ └── common.json

│ │ ├── de/

│ │ │ └── common.json

│ │ └── (other languages...)

│ ├── utils/

│ │ ├── unitConversion.ts

│ │ ├── datasheetExport.ts

│ │ ├── dateFormatter.ts

│ ├── backend/

│ │ ├── database/

│ │ ├── routes/

│ │ ├── middlewares/

│ ├── i18n.ts ✅ NEW - i18next configuration file

│ ├── types/

│ │ ├── datasheet.ts

│ │ ├── template.ts

│ │ ├── user.ts

│ │ ├── (all other types)

│ ├── validation/

│ │ ├── datasheetTemplateSchema.ts

│ │ ├── userSchema.ts

│ └── pages/ (optional for legacy routes)

├── package.json

├── next.config.js

├── tailwind.config.js

└── tsconfig.json

**✅ Key new additions**

| **Folder/File** | **Purpose** |
| --- | --- |
| src/locales/ | All translations live here now (en/common.json, etc.) |
| src/i18n.ts | Your central react-i18next configuration |
| src/components/common/LanguageSwitcher.tsx | Dropdown or button to change current language |
| src/hooks/useUILanguage.ts | Your custom hook to replace getLabel() with t() |

👉 Everything else (your backend, datasheets, templates, exports, etc.) remains as is.

✅ This structure is **100% compatible with Next.js App Router + TypeScript + Tailwind + react-i18next**.  
✅ Used in many real world projects at big companies.

**✅ Benefits for you**

* Translations fully removed from DB
* All translations tracked + versioned in Git
* Easier for teams to collaborate on translations
* Smaller DB + fewer API routes
* Cleaner separation of backend ↔ frontend concerns
* Instant language switching (i18n.changeLanguage())

This would be a **world-class architecture** for SpecVerse v2.

// src/i18n.ts

import i18n from 'i18next';

import { initReactI18next } from 'react-i18next';

import LanguageDetector from 'i18next-browser-languagedetector';

// ✅ Import your translation files

import en from './locales/en/common.json';

import fr from './locales/fr/common.json';

// Add other languages as needed

// import de from './locales/de/common.json';

// import jp from './locales/jp/common.json';

i18n

.use(LanguageDetector) // ✅ auto-detect user language from browser/cookies

.use(initReactI18next) // ✅ bind react-i18next to i18next instance

.init({

resources: {

en: { translation: en },

fr: { translation: fr },

// Add more here

// de: { translation: de },

// jp: { translation: jp },

},

fallbackLng: 'en', // ✅ default to English if language not found

debug: false, // Set to true for dev debugging

interpolation: {

escapeValue: false, // ✅ react already escapes values

},

detection: {

order: ['querystring', 'cookie', 'localStorage', 'navigator', 'htmlTag'],

caches: ['localStorage', 'cookie'], // ✅ cache language for user

},

});

export default i18n;

// src/components/common/LanguageSwitcher.tsx

"use client";

import { useTranslation } from "react-i18next";

const languages = [

{ code: "en", label: "English", flag: "🇺🇸" },

{ code: "fr", label: "Français", flag: "🇫🇷" },

// ✅ Add more if needed:

// { code: "de", label: "Deutsch", flag: "🇩🇪" },

// { code: "es", label: "Español", flag: "🇪🇸" },

];

export default function LanguageSwitcher() {

const { i18n } = useTranslation();

const currentLang = i18n.language;

return (

<div className="flex items-center gap-2">

{languages.map((lang) => (

<button

key={lang.code}

onClick={() => i18n.changeLanguage(lang.code)}

className={`px-3 py-1 rounded-md text-sm font-medium transition ${

currentLang === lang.code

? "bg-blue-600 text-white"

: "bg-gray-200 text-gray-800 dark:bg-gray-700 dark:text-gray-200"

} hover:bg-blue-500 hover:text-white`}

title={lang.label}

>

<span className="text-lg">{lang.flag}</span>

</button>

))}

</div>

);

}

**🎯 Recommended Final Backend Structure for SpecVerse**

pgsql

CopyEdit

src/

├── backend/

│ ├── config/ → SQL Server pool, environment config, etc.

│ │ └── db.ts

│ ├── controllers/ → Request handlers (business logic layer)

│ │ ├── datasheetController.ts

│ │ ├── clientController.ts

│ │ └── userController.ts

│ ├── database/ → Raw DB queries + reusable data access logic

│ │ ├── datasheetQueries.ts

│ │ ├── templateWriteQueries.ts

│ │ ├── duplicateSheet.ts

│ │ ├── clientQueries.ts

│ │ └── utils (optional folder for query helpers)

│ ├── middleware/ → Express middlewares (auth, logging, error handling etc.)

│ │ └── authMiddleware.ts

│ ├── routes/ → All Express routes

│ │ ├── datasheetsRoutes.ts

│ │ ├── clientsRoutes.ts

│ │ ├── categoriesRoutes.ts

│ │ ├── usersRoutes.ts

│ │ ├── languageRoutes.ts

│ │ └── labelRoutes.ts

│ ├── services/ → Non-DB services (PDF export, Excel export etc.)

│ │ ├── exportService.ts

│ │ └── unitConversionService.ts

│ ├── types/ → Shared backend types (optional but highly recommended)

│ │ └── datasheetTypes.ts

│ ├── utils/ → Miscellaneous utilities (formatting, date utils etc.)

│ │ └── logger.ts

│ └── server.ts → 🚀 Entry point: starts Express server

│

├── types/ → Shared types between frontend + backend (optional)

│ └── datasheetTemplate.ts

This is **very close to what you already have** 👍  
You only need small adjustments + cleanup + consistent folder purposes.

**✅ Key principles for this structure**

| **Folder** | **Purpose** |
| --- | --- |
| config/ | global settings, database pool |
| controllers/ | no SQL here, only req → res logic |
| database/ | all SQL queries, reusable logic |
| middleware/ | custom middlewares (auth, validation etc.) |
| routes/ | all Express routes, clean & modular |
| services/ | heavy tasks (PDF, Excel, etc.) |
| types/ | backend types for safety + documentation |
| utils/ | any utility/helper functions |
| server.ts | only starts the app, no logic inside |

This will scale very well even if SpecVerse grows to 100+ features.

**📝 Extra suggestions (for stability)**

1️⃣ Put **/api as prefix** for all backend routes:

ts

CopyEdit

app.use("/api/datasheets", datasheetsRoutes)

2️⃣ Optional: Add global **errorMiddleware**:

ts

CopyEdit

app.use(errorHandler)

3️⃣ Optional: Add **logger utility** in utils/logger.ts:

ts

CopyEdit

export const log = (...args: any[]) => console.log("[SpecVerse]", ...args)

4️⃣ Optional: Add **CORS + Helmet security** for production:

ts

CopyEdit

app.use(cors())

app.use(helmet())

**✅ Result: Final API endpoint examples**

ruby

CopyEdit

GET /api/datasheets/

POST /api/datasheets/:id/update

POST /api/datasheets/templates/:templateId/create-filled

GET /api/datasheets/:id/export/pdf

GET /api/datasheets/:id/subsheets

etc...

Very clean. Very future-proof.

**✅ Recommended Starter Folders (src/backend/)**

pgsql

CopyEdit

src/

├── backend/

│ ├── config/

│ │ └── db.ts → SQL Server connection

│ ├── controllers/

│ │ ├── datasheetController.ts

│ │ ├── clientController.ts

│ │ └── userController.ts

│ ├── database/

│ │ ├── datasheetQueries.ts

│ │ ├── templateWriteQueries.ts

│ │ ├── duplicateSheet.ts

│ │ └── clientQueries.ts

│ ├── middleware/

│ │ ├── authMiddleware.ts

│ │ └── errorHandler.ts → (optional global error middleware)

│ ├── routes/

│ │ ├── datasheetsRoutes.ts

│ │ ├── clientsRoutes.ts

│ │ ├── categoriesRoutes.ts

│ │ ├── usersRoutes.ts

│ │ ├── languageRoutes.ts

│ │ └── labelRoutes.ts

│ ├── services/

│ │ ├── exportService.ts

│ │ └── unitConversionService.ts

│ ├── types/

│ │ └── datasheetTypes.ts

│ ├── utils/

│ │ └── logger.ts

│ └── server.ts → 🎉 main entry point

This is exactly what **top real-world SaaS backends** look like.  
You can just copy this structure and grow SpecVerse safely.

**📝 Extra Tips**

* Only server.ts should start Express.
* All logic goes inside controllers/ + database/.
* Never write raw SQL in routes/. Only call helpers from database/.
* Add src/backend/.env support later if you want database URLs configurable.

**✅ src/backend/ folder structure**

**css**

**CopyEdit**

**src/backend/**

**│**

**├── config/**

**│ └── db.ts**

**│**

**├── controllers/**

**│ ├── datasheetController.ts**

**│ ├── clientController.ts**

**│ ├── categoryController.ts**

**│ ├── userController.ts**

**│ ├── languageController.ts**

**│ └── labelController.ts**

**│**

**├── database/**

**│ ├── datasheetQueries.ts**

**│ ├── templateWriteQueries.ts**

**│ ├── duplicateSheet.ts**

**│ ├── templateReferenceQueries.ts**

**│ ├── clientQueries.ts**

**│ ├── categoryQueries.ts**

**│ ├── userQueries.ts**

**│ └── languageQueries.ts**

**│**

**├── routes/**

**│ ├── datasheetsRoutes.ts**

**│ ├── clientsRoutes.ts**

**│ ├── categoriesRoutes.ts**

**│ ├── usersRoutes.ts**

**│ ├── languagesRoutes.ts**

**│ └── labelRoutes.ts**

**│**

**├── services/**

**│ ├── exportService.ts**

**│ ├── unitConversionService.ts**

**│ └── pdfGenerator.ts**

**│**

**├── utils/**

**│ ├── labelTranslation.ts**

**│ └── validationUtils.ts**

**│**

**└── server.ts ✅ (your main backend entry)**

**✅ Explanation**

| **Folder** | **Purpose** |
| --- | --- |
| **config/** | **DB connection + any env config.** |
| **controllers/** | **Handles logic for routes.** |
| **database/** | **Your SQL query files (all DB logic).** |
| **routes/** | **Your Express routers per feature.** |
| **services/** | **Business logic (PDF export, unit conversions, etc).** |
| **utils/** | **General helper functions, validations, etc.** |
| **server.ts** | **Your backend entry file.** |

**✅ Optional (recommended) additional files**

If you want to go full enterprise style:

CopyEdit

│ ├── models/

│ │ ├── datasheetModel.ts

│ │ └── userModel.ts

│ ├── validations/

│ │ ├── datasheetValidation.ts

│ │ └── userValidation.ts

👉 These would use your **Zod schemas** or future **Prisma models**.

**✅ Total file count: ~25**

This is a **perfect professional backend structure** for SpecVerse.  
It separates:

* DB config
* Controllers (logic layer)
* Routes (only routing)
* Services (reusable helpers)
* Middleware
* Utilities
* Type definitions
* Centralized server start

This structure will **scale very safely** even if SpecVerse becomes very big later.

**Additional Recommendations**

* **Loading State**: Implement a loading.tsx file in the same directory to show a loading indicator while the page is being generated.
* **Error Page**: Create an error.tsx file to handle unexpected errors gracefully.
* **Styling**: Utilize Tailwind CSS or another styling framework to enhance the appearance of your datasheet pages.
* **Content Sanitization**: Ensure that the HTML content rendered with dangerouslySetInnerHTML is sanitized to prevent security vulnerabilities.

**You are now getting exactly the same data-entry experience as Plant 3D, Aveva, Hexagon SmartPlant.  
This is what separates serious software from hobby code**

**📝 Next Phase Scope:**  
We are now proceeding with the following **new modules**:

1. **Project Estimation**
   * Dashboard for active/in-progress estimates
   * Create new estimation (based on datasheet references or manual input)
   * View past estimates history
2. **Inventory Management**
   * Stock levels (parts, materials, equipment)
   * Maintenance logs for equipment/assets
   * Integration with datasheets (optional)
3. **Remaining Multi-language Features**
   * Fully translate Project Estimation and Inventory modules
   * Complete any missing datasheet multi-language (e.g. project names, client names, etc.)
4. **Role-Based Permissions**
   * Define role permissions for:
     + Estimation module
     + Inventory module
     + Admin settings for multi-language entries
     + View-only vs edit vs admin control
5. **Audit Logs**
   * Record all changes/actions for:
     + Project estimates (creation, edits, approval)
     + Inventory stock changes (add, update, remove, maintenance logs)
     + User actions in these modules

**✅ 2. Suggested Modules / Features for INVENTORY MANAGEMENT**

**🎯 Inventory Dashboard**

* Summary of total items, low stock alerts (QuantityOnHand < ReorderLevel), maintenance due.
* Search & filter (by name, supplier, category).

**🎯 Inventory CRUD**

* Add/Edit/Delete inventory items.
* QuantityOnHand is read-only, controlled only via InventoryTransactions.

**🎯 Stock Transactions**

* Record Receive / Issue / Adjustment.
* Update QuantityOnHand accordingly.
* Show transaction history per item.

**🎯 Maintenance Logs**

* Add maintenance records to items.
* View full maintenance history per item.

**🎯 Permissions**

* Only roles with proper rights can:
  + View vs manage inventory
  + Log maintenance
  + Do stock adjustments

**🎯 Audit Logs**

* Track every insert/update/delete for items & logs.
* Show audit history per item.

**✅ 1. Suggested Folder Structure**

pgsql

CopyEdit

src/

├── backend/

│ ├── database/

│ │ ├── inventoryQueries.ts → Main SQL query functions

│ │ ├── inventoryTransactionQueries.ts

│ │ ├── inventoryMaintenanceQueries.ts

│ │ └── inventoryAuditQueries.ts

│ ├── routes/

│ │ └── inventoryRoutes.ts → All /api/backend/inventory/ routes

│ ├── services/

│ │ └── inventoryService.ts → Any business logic helpers

* You can optionally split further by responsibility, but this is enough for now.

**✅ 2. Recommended API Routes (inside inventoryRoutes.ts)**

| **Method** | **Route** | **Description** |
| --- | --- | --- |
| GET | /api/backend/inventory/ | Get all inventory items |
| GET | /api/backend/inventory/:id | Get single item + details |
| POST | /api/backend/inventory/ | Create new inventory item |
| PUT | /api/backend/inventory/:id | Update inventory item |
| DELETE | /api/backend/inventory/:id | Delete inventory item (soft or hard) |
| GET | /api/backend/inventory/:id/transactions | Get stock transactions for item |
| POST | /api/backend/inventory/:id/transactions | Add stock transaction |
| GET | /api/backend/inventory/:id/maintenance | Get maintenance logs for item |
| POST | /api/backend/inventory/:id/maintenance | Add maintenance log |
| GET | /api/backend/inventory/:id/audit | Get audit logs for item |
| GET | /api/backend/inventory-translations/:id/:lang | Get item translations |
| POST | /api/backend/inventory-translations/:id/:lang | Update item translation |

**✅ 3. Responsibilities of Query Files**

**inventoryQueries.ts**

* CRUD for InventoryItems
* Get items by filters (low stock, active only, etc.)

**inventoryTransactionQueries.ts**

* CRUD for InventoryTransactions
* Automatically update QuantityOnHand after receive/issue/adjust

**inventoryMaintenanceQueries.ts**

* CRUD for InventoryMaintenanceLogs

**inventoryAuditQueries.ts**

* Insert & read audit logs from InventoryAuditLogs

**✅ 4. inventoryService.ts (optional but recommended)**

* Wrap complex logic:
  + Example: addInventoryTransaction() → insert transaction + update stock + add audit log
  + Example: deleteInventoryItem() → check dependencies + soft delete + add audit log

**✅ 5. Middleware Considerations**

You can reuse your existing middleware:

* checkRolePermission() → restrict access by role
* logAudit() → central function to insert to InventoryAuditLogs

Example permission names:

nginx

CopyEdit

INVENTORY\_VIEW

INVENTORY\_CREATE

INVENTORY\_EDIT

INVENTORY\_DELETE

INVENTORY\_TRANSACTION\_CREATE

INVENTORY\_MAINTENANCE\_CREATE

This structure will **cleanly integrate into SpecVerse** without disrupting existing modules.  
It follows your current design principles:

* SQL-first backend logic (\*.Queries.ts)
* Clean route + service separation
* Full audit + role control

**✅ 1. Suggested Page Routes**

bash

CopyEdit

src/

├── app/

│ ├── (admin)/

│ │ ├── inventory/

│ │ │ ├── page.tsx → Inventory Dashboard

│ │ │ ├── create/page.tsx → Create new item

│ │ │ ├── [id]/page.tsx → View item details + tabs

│ │ │ ├── [id]/edit/page.tsx → Edit item

You can also optionally merge create and edit into a single dynamic component.

**✅ 2. Inventory Dashboard (/inventory)**

* Search + filters (by name, category, supplier)
* Table list of inventory items:
  + Item Code
  + Item Name
  + Quantity on Hand
  + Reorder Level (show warning badge if low stock)
  + Location
  + Actions: View | Edit | Delete

Component ideas:

php-template

CopyEdit

<InventoryListTable />

<InventoryFilters />

<LowStockBadge />

**✅ 3. Inventory Item Details (/inventory/[id])**

**Show as Tabbed View:**

css

CopyEdit

[ Overview | Transactions | Maintenance Logs | Audit Logs ]

**➡️ Overview Tab**

* Item details
* Current stock level
* View Translations per language

**➡️ Transactions Tab**

* List of stock changes
* Add Receive / Issue / Adjustment

**➡️ Maintenance Logs Tab**

* List of maintenance history
* Add new maintenance log

**➡️ Audit Logs Tab**

* Read-only list of changes for this item

Components:

php-template

CopyEdit

<InventoryItemHeader />

<InventoryItemDetails />

<StockTransactionTable />

<MaintenanceLogTable />

<AuditLogTable />

**✅ 4. Inventory Create / Edit Page**

Form with validation:

scss

CopyEdit

Item Code

Item Name

Description

Category (dropdown)

Supplier (dropdown)

Manufacturer (dropdown)

Location

Reorder Level

UOM

* Use shared Zod schema for validation
* QuantityOnHand is system-managed, not editable from this form.

Reusable component:

arduino

CopyEdit

<InventoryForm isEditing={true/false} />

**✅ 5. Role Permissions (UI level)**

* checkUserPermission() helper (like your datasheet system) to conditionally show:
  + Create button
  + Edit/Delete buttons
  + Add Transaction button
  + Add Maintenance button

Example:

tsx

CopyEdit

{canUser('INVENTORY\_CREATE') && (

<Button onClick={() => router.push('/inventory/create')}>Add New Item</Button>

)}

**✅ 6. Multi-language integration**

* Add Translations section in /inventory/[id] → Overview Tab:
  + Dropdown to select language
  + Show ItemNameTranslation, DescriptionTranslation
  + Inline edit (for users with permission)

Example:

php-template

CopyEdit

<InventoryTranslationsEditor inventoryId={id} />

This reuses your existing UILabelTranslations pattern.

**✅ Optional: Toasts & Notifications**

* Show success/failure toasts:
  + After create/update/delete item
  + After stock transactions
  + After maintenance logs

Example:

scss

CopyEdit

showToast("Inventory item successfully updated");

**✅ Summary: Page + Component Flow**

| **Page** | **Key Components** |
| --- | --- |
| /inventory | InventoryListTable, InventoryFilters |
| /inventory/create | InventoryForm |
| /inventory/[id] | InventoryItemHeader, Tabbed View (Overview, Transactions, Maintenance, Audit) |
| /inventory/[id]/edit | InventoryForm |
| Shared | LowStockBadge, AuditLogTable, TranslationsEditor |

This design will **fit naturally into your current SpecVerse UI & logic**, just like datasheets and templates.

You can fully reuse:

* Zod validation patterns
* checkUserPermission()
* Tailwind + Heroicons + shadcn/ui components