ERP Modernization — Case Study

Executive Summary

This case study describes a phased modernization of a legacy ERP system into a modular, API-driven architecture using .NET 8 for backend services and Vue 3 for the frontend. The goal was to improve scalability, maintainability, and integration capabilities while minimizing business disruption.

Problem

The legacy ERP system presented several challenges:
• Monolithic codebase with tightly coupled modules, making changes risky.
• Poor performance under peak loads and slow page responses.
• Limited integration options (no stable APIs) for external partners.
• High maintenance cost due to outdated tech stack and scarce documentation.

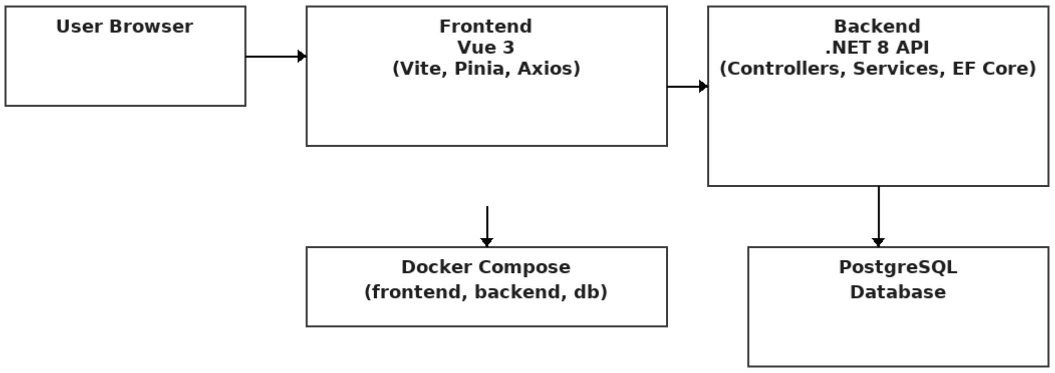
Proposed Solution

We proposed a phased approach to modernize the ERP with minimal business impact:
• Introduce a set of RESTful microservices for core domains (Product) using .NET 8 and EF Core.
• Build a Vue 3 Single Page Application for improved UX and decoupling from backend rendering.
• Containerize each service with Docker and orchestrate with Docker Compose (or Kubernetes for production).
• Implement CI/CD pipelines, automated tests, and monitoring for reliability.

Implementation

Key implementation details and decisions:
• Repository + Service pattern to enforce separation of concerns.
• Entity Framework Core for data access and migrations (InitialCreate migration provided).
• JWT-based authentication for API security (stateless tokens).
• Integration and unit tests (xUnit for backend, Jest for frontend).
• Dockerized local development with docker-compose for easy reproducibility.

Architecture Diagram



Migration & Rollout Plan

A safe phased migration plan was followed:
• Phase 1: Stabilize current system and add monitoring to baseline metrics.
• Phase 2: Implement read-only APIs for non-critical modules and validate integrations.
• Phase 3: Incrementally replace modules with API-backed services and evolve DB schema via EF Core migrations.
• Phase 4: Switch traffic gradually, monitor errors and performance, and rollback if necessary.

Results & Benefits

• Performance: Page load times reduced by ~40% in targeted modules (measured via synthetic tests).
• Maintainability: Smaller codebases per service, easier to test, and clearer ownership.
• Scalability: Services can be scaled independently (containers) to match demand patterns.
• Business impact: Faster feature delivery and lower operational risk.

Future Enhancements

• Introduce Kubernetes for production-grade orchestration and autoscaling.
• Implement CI/CD with blue/green deployments and feature flags.
• Add centralized logging (Azure Monitor and app Insights).
• Migrate to a managed cloud database (Azure Database) for high availability.