VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

SOFTWARE ENGINEERING STUDY PROGRAMME

PS Software Design

PS Sofware Design

Gustas Mickus, Julius Jauga, Tadas Riksas, Darius Spruogis

Supervisor : Vasilij Savin

Vilnius
2025

Summary

A brief overview of how our implementation differs from the architecture proposed in the reference document.

Contents

1 Changes

1.1 Single UI application

Original Specification: Three UIs: Web POS, Admin Console, Support Console.
Actual Implementation: Single React application.
Why:

- Maintaining three separate frontend applications creates additional deployment and maintenance complexity.
- Easier to share components and state across all views.
- Role-based rendering within single app is easier to implement for the small team.

1.2 Computer-Focused UI Design

Original Assumption: Mobile-first (vertical) UI.
Actual Implementation: Desktop/laptop–focused(horizontal) UI.
Why:

- Restaurant, spa, etc. management tasks involve data tables that are inefficient, harder to use on small vertical screens.
- Primary usage occurs during business hours at fixed locations (front desk, office, back counter) where computers are already available.
- A computer-focused UI reduces development complexity by avoiding mobile-specific flows.

1.3 No BFF / API Gateway

Original Assumption: Use a Backend-for-Frontend (BFF) or API Gateway layer.
Actual Implementation: Simple reverse proxy using Nginx.
Why:

- Implementing a BFF or API Gateway introduces additional services, contracts, and deployment complexity that exceeded the available development time.
- The system serves a single UI application, making a BFF layer unnecessary.
- Reverse proxy(Nginx) provides sufficient routing, security.

1.4 Single backend application

Original Assumption: Multiple services?
Actual Implementation: A single modular monolith backend applicaton.
Why:

- A single backend simplifies development, testing, and debugging for small team.
- The backend encapsulates the *Domain Core* through domain-driven modules (`src/modules`), making a separate domain service unnecessary.
- Because logic frequently depends on TypeORM entities, repositories, and external services (Stripe, etc.), the domain is intentionally coupled to infrastructure. That does make future large-scale refactors harder. Nevertheless, for a small team with limited development time and a need for simplicity, this trade-off is acceptable since it reduces boilerplate and cognitive overhead.

1.5 Authentication

Original Assumption: Username + password, no additional details given on the mechanism
Actual Implementation: Email + password using JWT (JSON Web Token) access token + refresh token in HTTP-only cookie.
Why:

- HTTP-only cookies prevent JavaScript on the client from accessing refresh tokens making it more more secure.
- JWTs allow the backend to operate without maintaining session state in memory.
- Email is a unique global professional identifier, whereas ”usernames” do not really fit in business application.

1.6 Super admin creation

Original Assumption: No details given on how the initial system root user is created.
Actual Implementation: Super admin is created using a SQL script via TypeORM directly in the database.
Why:

- Simple and removes the need for a public-facing registration endpoint for the most powerful role in the system.

1.7 Business creation management

Original Assumption: Implied that the super admin creates businesses.

Actual Implementation: Super admin explicitly creates the Business alongside the initial Owner account.

Why:

- A business entity cannot exist without an owner, and an owner cannot exist without a business.
- No need for registration page.
- The owner can later manage users and permissions for that business.

1.8 Permission & Role management

Original Assumption: Static, hardcoded role system consisting of three specific levels: Owner, Employee, and Admin.

Actual Implementation: Initial super admin role (app global). Additionally, business owners can create custom roles assigned with preset permissions (e.g., INVENTORY, CATEGORY, ORDERS, APPOINTMENTS...).

Why:

- Hard coded roles may be too limiting for some business, allows more flexibility.
- Allows owners to grant least privileges needed for employee.

1.9 Inventory management

Original Assumption: Not specified.

Actual Implementation: Inventory is tracked via ”Product” (with units like kg/liters) and ”Stock Change” logs (supply/usage/waste/adjustment) and ”Stock Level” to store total quantities.

Why:

- Allows tracking raw ingredients (e.g., flour in kg).
- Stock changes are recorded as entries (Supply vs. Waste), providing a history of stock levels changes, rather than just overwriting a number. Stock level table is used to store total quantities.

1.10 Menu item management

Original Assumption: A ”Product” is something the business sells, implying a 1:1 relationship between what is sold and what is in inventory.

Actual Implementation: ”Menu Item” is a separate entity from ”Inventory Product”. A Menu Item can have variations and uses recipes to link to inventory products.

Why:

- Businesses sell ”Burgers” (Menu Item) but stock ”Buns” and ”Meat” (Inventory). A 1:1 mapping is not enough when also implementing inventory .
- Allows a single Menu Item (e.g., Latte) to have variations (Small/Large) that consume different amounts of inventory.

1.11 Appointment management

Original Assumption: Generic ”Service” checking availability against a pool of employees.

Actual Implementation: ”Service Definition” (e.g., Haircut) exists separately from ”Staff Service” (Employee X can do Haircut). Appointments link Customer, Staff Service, and Time, validated against employee configurable weekly availability.

Why:

- Not every employee performs every service.

1.12 Order management

Original Assumption: A single ”Order” table handles both restaurant items and service reservations.

Actual Implementation: Restaurant orders are represented with ‘Order‘, ‘OrderPayment‘, and ‘OrderItem‘ tables. Service appointments are represented separately with ‘Appointment‘, ‘AppointmentPayment‘, and ‘AppointmentPaymentItem‘ tables.

Why:

- Service bookings (time-based, specific staff) have different attributes than restaurant sales (inventory-based).
- Separating them avoids a huge single table with many null values (e.g., an order for a coffee doesn’t need a ”start_time” or ”employee_id”).

1.13 Payment management

Original Assumption: Simple choice of Cash, Card, or Gift Card.

Actual Implementation: Logic splits payments into Gift Cards (valid for items/services only, not tips) and then Cash. If paying by card (Stripe), the UI first applies any gift card, then creates a Stripe PaymentIntent only for the remaining balance.

Why:

- Gift cards usually cannot be used to pay out tips (which go to employees).
- Gift card may not cover full payment, so gift cards(if applied) are processed first to determine the remaining balance for Cash/Stripe.

1.14 Stripe Payment Integration

Original Assumption: The specification diagram showed a simplified synchronous flow with direct ”payment confirmation” between Payment Provider and System, suggesting immediate processing.

Actual Implementation: Asynchronous webhook-based confirmation:

1. Frontend initiates Stripe Elements Checkout (in our case, an embedded card form) with PaymentIntent client secret
2. Stripe confirms payment client-side
3. Backend receives webhook event (not client response)
4. Webhook handler verifies signature, extracts metadata, and fulfills payment
5. Frontend polls order status and closes modal once confirmed

Why:

- Webhooks ensure payment processing even if the client crashes, loses connection, or closes the browser post-payment
- Backend trusts Stripe’s authenticated webhook rather than client confirmation, preventing lost transactions
- Frontend polling provides UX feedback during asynchronous processing

1.15 Discount Management

Original Assumption: Discounts can be applied at both product (menu item/service) level and order level, with each discount selected independently by the employee.

Actual Implementation: Discounts follow a strict priority hierarchy and are applied automatically without employee selection:

1. **Order-level Discount (Priority 1):** If an active order-level discount exists, it is automatically applied to the entire order total and *takes priority* over all item-level discounts.
2. **Menu Item/Service-level Discounts (Priority 2):** Only if no order-level discount exists, individual item-level discounts are summed and applied to their respective items.
3. **Manual Discounts:** Employees can still apply an additional manual discount by entering a dollar amount, which is tracked separately from automatic discounts.

Why:

- Order-level discounts should take precedence and prevent stacking of multiple item-level discounts, which could result in over-discounting.

- Automatic discount calculation eliminates employee error.
- Separating automatic and manual discounts preserves audit trails: the system records what promotion was applied versus what the employee manually adjusted.

1.16 Audit logging

Original Assumption: A single generic AuditLog table capturing ”things like order status changes or role updates”, storing ‘before_json‘ and ‘after_json‘.

Actual Implementation: Split logging into Business Audits (saves data changes via service wrappers) and Security Audits (login attempts, IP addresses).

Why:

- Security logs (authentication events) may have different retention and privacy requirements than Business logs (operational changes).

2 General feedback

2.1 Evaluation Breakdown

Category	Assessment	Score
Data Model Specification	We had to add way more tables and different columns than provided. Some things were just missing.	5/10
Business Flow Specification	Good diagrams for main flows (orders, payments, reservations). Lacked details on super admin workflows, business creation.	7/10
API Specification	General guidance provided but lacked detailed contracts, schemas, and error codes. Teams had to make many implementation decisions independently.	6/10
Architecture Specification	High-level layers defined but overly complex for small team (three UIs, BFF layer, multiple services). Should have been more pragmatic given team size and timeline.	6/10
Feature Completeness Specification	Core features well-described. Missing clarity on inventory approach, authentication details, permission model, gift cards, and SMS.	6/10
Integration Specification	External services mentioned but integration patterns not specified. Missing Stripe approach, webhook details, SMS provider guidance, and error handling.	6/10
Error Handling & Edge Cases	Focused on happy paths. Did not address payment failures, inventory shortages.	6/10
UI/UX Specification	Mockups communicated flows well but mobile-first assumption was impractical. Business management with data tables needs desktop interface, especially given fixed location usage.	6/10

1 table. *Specification Quality Assessment*

2.2 Overall Evaluation

Overall Score: 6 / 10.0

This evaluation reflects what our team thinks about the document that other team provided, of course the team did put in effort so that has to be taken into consideration.