CADT
IDT
វិទ្យាស្ថានបច្ចេកវិទ្យាឌីជីថល
Institute of Digital Technology

ក្រសួងប្រៃសណីយ៍និងទូរគមនាគមន៍
MINISTRY OF POST AND TELECOMMUNICATIONS

Capstone 1 Project Report on

Modula Point of Sale (POS) System

at

Cambodia Academy of Digital Technology

Submitted By:

Mr. Lim Ieangzorng

Ms. Chhun Nika

Ms. Kon Sotheara

Mr. Tong Vorak

Mr. Rith Seyhak

Mr. Chey Rotana

Under the Advisory of:

Mr. Thear Sophal

Date of Defense: December 24th, 2025

This report is submitted in partial fulfillment of the requirements for the Bachelor of Computer Science degree with a specialization in Software Engineering

Department of Computer Science

Faculty of Digital Engineering

បណ្ឌិត្យសភាបច្ចេកវិទ្យាឌីជីថលកម្ពុជា
វិទ្យាស្ថានបច្ចេកវិទ្យាឌីជីថល
មហាវិទ្យាល័យ វិស្វកម្មឌីជីថល
ដេប៉ាតឺម៉ង់ វិទ្យាសាស្ត្រកុំព្យូទ័រ

គម្រោងសញ្ញាបត្របរិញ្ញាបត្រ

របស់ក្រុម ៖ ២

កាលបរិច្ឆេទនិក្ខេបបទ ៖ ថ្ងៃទី ២៤ ខែ ធ្នូ ឆ្នាំ ២០២៥

អនុញ្ញាតអោយការពារគម្រោង ៖

ប្រធានបណ្ឌិត្យសភាបច្ចេកវិទ្យាឌីជីថលកម្ពុជា_____

រាជធានីភ្នំពេញ ថ្ងៃទី __ ខែ _____ ឆ្នាំ _____

ប្រធានបទ ៖ Modula Point of Sale (POS) System

សហគ្រាស ៖ បណ្ឌិតសភាបច្ចេកវិទ្យាឌីជីថលកម្ពុជា

ក្រូដឹកនាំគម្រោង ៖ លោក ធៀ សុផល _____

Cambodia Academy of Digital Technology

Institute of Digital Technology

Faculty of Digital Engineering

Department of Computer Science

CAPSTONE REPORT OF

Group : 2

Defense Date : December 24th, 2025

Capstone I Defense Authorization

President : _____

Phnom Penh,

Topic : Modula Point of Sale (POS) System

Establishment :

Advisor : Mr. Thear Sophal _____

ACKNOWLEDGMENT

The project team would like to express sincere gratitude to H.E. Dr. Seng Sopheap, President of the Cambodia Academy of Digital Technology (CADT), for his leadership, vision, and commitment to fostering an environment that supports innovation, research, and academic excellence. His guidance and support have contributed significantly to the learning opportunities that made this project possible.

The project team would also like to extend heartfelt appreciation to Mr.Thear Sophal, the project advisor, for their invaluable guidance, continuous support, and constructive feedback throughout the development of this capstone project. Their expertise, patience, and insightful suggestions significantly contributed to the quality of the system design, implementation, and documentation.

In addition, sincere thanks are extended to the faculty members of CADT for their academic guidance, encouragement, and knowledge shared throughout the program. The project team is also grateful to the Cambodia Academy of Digital Technology for providing the necessary resources, facilities, and supportive learning environment that made the completion of this project possible.

ABSTRACT

Point of Sale (POS) systems are essential tools for small and medium-sized businesses, particularly in food and beverage environments where accuracy, efficiency, and operational control are critical. Many existing POS solutions, however, present high entry costs, rigid hardware requirements, or limited flexibility, making them unsuitable for businesses that grow incrementally. This capstone project presents **Modula**, a modular and device-agnostic POS system designed to provide a low entry point while supporting gradual feature expansion as business needs evolve.

Modula is implemented as a high-fidelity web-based prototype optimized for mobile devices, with an initial focus on food and beverage operations. The system adopts a modular monolithic architecture, separating core responsibilities such as authentication and authorization, tenant and branch context, policy configuration, audit logging, and offline synchronization from feature modules including sales and order management, menu and inventory management, cash sessions, staff management, attendance tracking, reporting, and discounts. This structure enables maintainability and controlled evolution without the complexity of microservices.

A key outcome of the project is the iterative refinement of module boundaries and specifications through continuous feedback from frontend development, backend integration, and manual usage testing. The resulting prototype demonstrates realistic POS workflows and highlights practical challenges in modular system design. Modula illustrates how a carefully structured architecture can balance usability, scalability, and feasibility within an academic capstone scope.

មូលនិយសង្ខេប

ប្រព័ន្ធចំណុចលក់(Point of Sale POS)គឺជាឧបករណ៍ដ៏សំខាន់សម្រាប់អាជីវកម្មខ្នាតតូច និងមធ្យម ជាពិសេសក្នុងវិស័យម្ហូបអាហារ និងភេសជ្ជៈ ដែលតម្រូវឱ្យមានភាពត្រឹមត្រូវ ប្រសិទ្ធភាព និងការគ្រប់គ្រងប្រតិបត្តិការយ៉ាងម៉ត់ចត់។ ទោះជាយ៉ាងណា ដំណោះស្រាយ POS ដែលមានស្រាប់ជាច្រើន មានតម្លៃថ្លៃក្នុងការដាក់ប្រើប្រាស់ ទាមទារអោយទិញឧបករណ៍ជាមុន ឬមានភាពបត់បែនតិចតួច ដែលធ្វើឱ្យវាមិនសមស្របសម្រាប់អាជីវកម្មដែលរីកចម្រើនជាបន្តបន្ទាប់។ គម្រោង Capstone នេះបង្ហាញអំពី Modula ដែលជាប្រព័ន្ធ POS មានលក្ខណៈម៉ូឌុល និងមិនពឹងផ្អែកលើឧបករណ៍ជាក់លាក់ណាមួយ ត្រូវបានរចនាឡើងដើម្បីផ្តល់ភាពងាយស្រួលក្នុងការចាប់ផ្តើមប្រើប្រាស់ និងអនុញ្ញាតឱ្យពង្រីកមុខងារបានជាបន្តបន្ទាប់ទៅតាមតម្រូវការអាជីវកម្ម។
Modula ត្រូវបានអភិវឌ្ឍជាគំរូគោលទំព័រដែលមានកម្រិតភាពជាក់លាក់ខ្ពស់ (high-fidelity prototype) និងត្រូវបានបង្កើនប្រសិទ្ធភាពសម្រាប់ការប្រើប្រាស់លើឧបករណ៍ចល័ត ដោយផ្តោតដំបូងទៅលើប្រតិបត្តិការវិស័យម្ហូបអាហារ និងភេសជ្ជៈ។ ប្រព័ន្ធនេះអនុវត្តស្ថាបត្យកម្មម៉ូណូលីតម៉ូឌុល (modular monolithic architecture) ដោយបំបែកទំនួលខុសត្រូវស្នូល ដូចជា ការផ្ទៀងផ្ទាត់ និងការអនុញ្ញាត (authentication and authorization) បរិបទអ្នកជួល និងសាខា ការកំណត់ច្បាប់គ្រប់គ្រងតាមតម្រូវការអ្នកជួល ការកត់ត្រាសកម្មភាព (audit logging) និងការធ្វើសមកាលកម្មក្រៅបណ្តាញ (offline synchronization) ចេញពីម៉ូឌុលមុខងារ ដូចជា ការលក់ និងការគ្រប់គ្រងការបញ្ជាទិញ ការគ្រប់គ្រងម៉ឺនុយ និងស្តុក ការគ្រប់គ្រងវគ្គប្រាក់ ការគ្រប់គ្រងបុគ្គលិក ការចុះវត្តមាន ការរាយការណ៍ និងការបញ្ជាះតម្លៃ។
រចនាសម្ព័ន្ធនេះអនុញ្ញាតឱ្យប្រព័ន្ធមានភាពងាយស្រួលក្នុងការថែទាំ និងការអភិវឌ្ឍបន្ត ដោយមិនបន្ថែមភាពស្មុគស្មាញដូចស្ថាបត្យកម្ម microservices។
លទ្ធផលសំខាន់មួយនៃគម្រោងនេះគឺការកែលម្អព្រំដែន
និងការពិពណ៌នាម៉ូឌុលជាបន្តបន្ទាប់ តាមរយៈមតិយោបល់ពីការអភិវឌ្ឍផ្នែក frontend ការរួមបញ្ចូលផ្នែក backend និងការសាកល្បងប្រើប្រាស់ដោយដៃ។ គំរូប្រព័ន្ធដែលទទួលបានបង្ហាញពីលំហូរការងារ POS ដែលជាក់ស្តែង និងបញ្ជាក់ពីបញ្ហាប្រឈមជាក់លាក់ក្នុងការរចនាប្រព័ន្ធមានលក្ខណៈម៉ូឌុល។ Modula បង្ហាញថា ស្ថាបត្យកម្មដែលត្រូវបានរៀបចំយ៉ាងម៉ត់ចត់ អាចធ្វើតុល្យភាពរវាងភាពងាយប្រើប្រាស់ សមត្ថភាពពង្រីក និងភាពអនុវត្តបានក្នុងបរិបទគម្រោង Capstone ផ្នែកសិក្សា។

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
F&B	Food and Beverage
HTTP	HyperText Transfer Proctocol
HTTPS	HyperText Transfer Proctocol Secure
ORM	Object-Relational Mapping
OTP	One Time Password
pnpm	Performant Node Package Manager
POS	Point of Sale
RBAC	Role Base Access Control
REST	Represational State Transfer

I. INTRODUCTION

This section provides an overview of our capstone project, which was designed to give us hands-on experience with practical problems by encouraging us to apply theoretical knowledge, technical skills, and business logic in a real-world context. Throughout this valuable period, we had the opportunity to contribute as a team and engage deeply with the project's objectives, gaining insights into effective project management, teamwork, and the application of software development principles. This experience has helped us understand the workflow, challenges, and solutions involved in developing a complete system, preparing us for future professional endeavors.

1.1. Presentation of Capstone

In our third year, we are required to complete a capstone project. This capstone project is an important part of our curriculum, designed to give us the opportunity to apply the knowledge and skills we have gained throughout our studies in a practical and meaningful way.

The purpose of the capstone project is to solve real-world problems by developing a complete and functional system. It challenges us to integrate theories and concepts from various subjects such as software engineering, system analysis, and project management. This experience prepares us for the challenges we may face in professional environments after graduation.

Throughout the project, we are expected to conduct research, analyze requirements, design the system, implement the solution, and report on our progress and results. This process allows us to demonstrate essential skills, including project management, teamwork, technical application, problem-solving, and effective communication.

By working on the capstone project as a team, we aim to gain hands-on experience in managing a real project from start to finish and to prove our ability to collaborate and deliver a quality system that meets specific needs.

1.1.1. Objective of Capstone

The main objective of our capstone project is to provide a comprehensive academic experience where we apply and integrate the knowledge and skills acquired throughout our degree program. This includes concepts from various subjects such as software engineering,

system analysis, database management, and project management. By combining these disciplines, we aim to demonstrate our ability to develop practical and effective solutions.

We aim to design and develop a functional system or solution that addresses a real-world problem. This process helps us improve our critical thinking and problem-solving skills by analyzing requirements, creating effective designs, and implementing reliable solutions.

Additionally, the project prepares us for real-world software development by simulating a professional environment where teamwork, communication, and time management are essential. Working as a team, we learn to coordinate tasks and combine our efforts to deliver a complete and well-documented product.

Overall, the capstone project is meant to build our confidence and readiness for future careers in software development by giving us practical experience in managing and completing a full project lifecycle.

1.1.2. Duration of Capstone

The capstone project is conducted over a total period of six months and is divided into two main stages: Capstone I and Capstone II. This report focuses exclusively on the activities carried out during Capstone I.

Capstone I is undertaken within the academic period from 1 October 2025 to 21 December 2025, spanning approximately 10 to 12 weeks. Unlike a rigid project timeline where all development phases follow a fixed schedule, the capstone project allows each team to define and manage its own workflow based on project scope and objectives.

For Capstone I, the primary focus is on the design phase of the project. The main objective during this stage is to analyze requirements, propose system architecture, and produce a functional prototype that demonstrates the core concepts and intended features of the system. Activities such as full-scale development, deployment, and extensive testing are deferred to Capstone II.

This flexible structure enables teams to allocate time effectively according to their project needs while ensuring that Capstone I delivers a well-defined system design and prototype that serves as a foundation for subsequent implementation in Capstone II.

II. PRESENTATION OF THE PROJECT

In this section, we will present an overview of the project we were assigned to and had the opportunity to work on as a team. We will explore the various aspects and perspectives that led us to initiate this project, including the problems we aimed to solve and the solutions we proposed. Additionally, we will explain how the project was planned from the outset, the team structure responsible for handling it, and the breakdown of tasks to ensure smooth progress.

2.1. Presentation of Capstone

Modula is a modular Point of Sale (POS) system designed to provide small and medium-sized businesses with a flexible and accessible digital solution for managing daily operations. The project was developed as part of Capstone I with the objective of addressing practical challenges faced by emerging businesses, particularly in the Food and Beverage (F&B) sector, where operational efficiency, cost sensitivity, and adaptability are critical factors for sustainability.

In its current scope, Modula focuses on delivering a web-based POS system optimized for mobile devices. This design choice reflects real-world usage patterns observed in small F&B businesses, where owners and staff frequently rely on smartphones or shared devices rather than dedicated POS terminals. By targeting the web platform first, Modula allows businesses to adopt digital operations without requiring immediate investment in specialized hardware, such as cash drawers or receipt printers. This approach lowers the entry barrier for digital transformation and enables businesses to experiment with structured operations before committing to additional costs.

A central concept behind Modula is its modular design philosophy. Instead of providing a monolithic system with a fixed and extensive feature set, Modula is structured around independent modules that can be enabled, configured, or extended as business needs evolve. This aligns with the principle of “pay only for what you need,” allowing businesses to start with essential functionality such as sales and inventory management and gradually scale up by adopting additional modules when required. In Capstone I, this modularity is expressed through clearly defined core and feature modules, each responsible for a specific domain of the system.

Although Modula is initially developed with the F&B industry as its primary use case, the system is intentionally designed as a foundational platform rather than an industry-specific endpoint. The architectural decisions made during Capstone I such as modular separation,

policy-driven behavior, and device-agnostic access are intended to support future adaptation to other business domains, including retail or service-oriented operations. However, it is important to emphasize that Capstone I does not aim to support multiple industries simultaneously. Instead, it establishes a solid and extensible foundation upon which future iterations of Modula can be built.

From a broader perspective, Modula represents an effort to bridge the gap between informal business practices and fully digital operations. Many small businesses operate without integrated systems, relying on manual tracking or disconnected tools that limit visibility and scalability. Modula seeks to address this gap by offering a POS system that is both technically robust and pragmatically aligned with the constraints of small business environments. By combining a low entry point, modular scalability, and a mobile-friendly web interface, Modula positions itself as a practical steppingstone for businesses transitioning toward more structured and data-driven operations.

In summary, Modula is presented in this capstone project as a modular, web-based POS system tailored to the needs of small F&B businesses. While limited in scope to ensure feasibility within academic and resource constraints, the system embodies design principles that support long-term growth, extensibility, and real-world applicability beyond the boundaries of Capstone I.

2.2. Problems

Small and medium-sized businesses, particularly in the Food and Beverage (F&B) sector, face a range of operational challenges when operating without an integrated digital Point-of-Sale (POS) system. These challenges are not limited to technical inefficiencies but extend to business sustainability, scalability, and competitiveness in increasingly digital markets. The absence of an appropriate POS solution often results in fragmented workflows, limited visibility into operations, and difficulties in adapting to business growth.

- One of the most significant problems is reliance on manual or semi-digital processes for handling sales, inventory, and staff activities. Many small businesses record transactions on paper, spreadsheets, or disconnected applications. While these methods may appear sufficient at an early stage, they are prone to human error, lack real-time insight, and make it difficult to trace historical data accurately. As transaction volume increases, these limitations become more pronounced, leading to inconsistencies in financial records and inventory counts.

- Another critical issue is **inventory mismanagement**. Without a system that systematically tracks stock movements, businesses struggle to maintain accurate inventory levels. Over-ordering results in waste and increased costs, while under-ordering leads to stock shortages and lost sales opportunities. In F&B environments, where ingredients may expire and menu items depend on precise stock availability, poor inventory tracking directly affects both operational efficiency and customer satisfaction.
- **Cost and accessibility** also present substantial barriers to digital adoption. Many commercial POS systems are designed with larger businesses in mind and require upfront investment in dedicated hardware such as terminals, receipt printers, and cash drawers. For small or newly established businesses, this financial commitment can be prohibitive. As a result, owners delay digital adoption or rely on improvised solutions that do not scale effectively. Even when software-only solutions exist, they often bundle unnecessary features, forcing businesses to pay for functionality they do not yet need.
- In addition, existing POS solutions frequently **lack flexibility** and **scalability**. Businesses that start small may later expand by adding staff, branches, or new operational workflows. Systems that are rigid in design make such transitions difficult, often requiring migration to entirely new platforms. This disrupts operations and increases the long-term cost of digitalization. Without a modular approach, businesses are forced into an all-or-nothing commitment rather than being able to grow incrementally.
- There is also a growing **competitive pressure** for businesses to adopt digital solutions. Customers increasingly expect faster service, clearer receipts, and consistent pricing, while business owners need access to operational data to make informed decisions. Businesses that continue to operate without structured digital systems risk being outperformed by competitors who leverage data-driven insights and streamlined workflows. In this context, the lack of an accessible POS system is not merely an operational inconvenience but a strategic disadvantage.
- Finally, many available solutions fail to account for **real-world usage constraints**, such as intermittent internet connectivity or shared device usage. In practical environments, especially in developing regions, network reliability cannot always be assumed. Systems that require constant connectivity or dedicated devices are poorly suited to such conditions, further limiting their usability for small businesses.

These problems collectively highlight the need for a POS system that is affordable, flexible, device-agnostic, and capable of supporting gradual business growth. Modula is proposed as a response to these challenges, aiming to provide a practical and modular solution that addresses operational inefficiencies while remaining aligned with the realities faced by small F&B businesses.

2.3. Target Users

Modula is designed as an organizational Point of Sale system intended to support multiple users operating within a single business or enterprise context. Rather than targeting individual consumers, Modula addresses the needs of businesses by accommodating different categories of users who interact with the system in distinct ways. These users differ not only in their responsibilities, but also in their relationship to the system as customers, operators, or employees. Clearly distinguishing these groups is essential to understanding Modula's design rationale and intended usage.

2.3.1. Business Owners (Primary Customers)

The primary target users of Modula are business owners, who represent the system's core customers. Business owners are responsible for subscribing to the platform, adopting it within their organization, and determining how it is deployed across their operations. In the context of Capstone I, Modula is initially positioned for small to medium-sized food and beverage businesses; however, its architectural design anticipates expansion into other industries in later stages.

Business owners may operate their businesses in different ways. Some owners manage a single outlet and personally handle daily operations, while others oversee multiple outlets and delegate operational responsibilities to staff. Modula is designed to support both scenarios by offering a low entry barrier for small operations and the ability to scale as the business grows. The "pay only for what you need" principle reflects this flexibility, allowing owners to start with minimal functionality and gradually adopt additional features or branches as required.

Importantly, business owners represent the organizational authority behind the system. Even when they do not interact with Modula on a daily basis, they remain responsible for defining the structure of the organization within the system, including assigning operational users and determining how the system supports business workflows.

2.3.2. Delegated Operators (Administrators and Managers)

A second group of target users consists of delegated operators, typically administrators and managers appointed by the business owner. These users operate within the organizational context of the business and act on behalf of the owner in managing day-to-day activities. They are not customers of Modula themselves; rather, their access to the system is granted through delegation.

Delegated operators are common in businesses where owners are not continuously present, such as multi-branch enterprises or operations with distributed management structures. These users are responsible for overseeing operational tasks, coordinating staff, and ensuring that daily activities run smoothly. Their role reflects a trust relationship with the business owner and highlights Modula's emphasis on organizational delegation rather than individual ownership.

By explicitly supporting delegated operators, Modula aligns with real-world business practices, where system usage is shared among multiple responsible parties rather than centralized around a single individual.

2.3.3. Frontline Staff (Cashiers and Operational Employees)

The third group of target users comprises frontline staff, such as cashiers and other operational employees. These users interact with Modula as part of routine service delivery and transactional workflows. Their engagement with the system is typically task-oriented and focused on operational efficiency.

Frontline staff use Modula to perform activities such as processing sales, handling orders, and supporting daily service operations. They do not participate in system configuration, organizational decisions, or subscription management. Their access exists strictly within the boundaries defined by the business owner and delegated operators.

This distinction reinforces Modula's design as an organizational system in which all users operate under a defined business structure. Frontline staff represent the most frequent users of the system, yet they remain functionally and structurally separate from ownership and management roles.

2.4. Objective

The development of Modula is guided by a set of clearly defined objectives that reflect both the academic goals of the Capstone I project and the broader vision of creating a practical, scalable Point of Sale system. These objectives are structured around a central main objective, supported by several specific objectives that address technical, operational, and contextual considerations.

2.4.1. Main Objective

The main objective of this project is to design and implement a modular, web-based Point of Sale system that supports essential business operations for small and medium-sized food and beverage enterprises, while remaining adaptable for future expansion into other industries.

Modula aims to demonstrate how a thoughtfully designed POS system can balance real-world usability with sound software engineering principles. Rather than delivering a feature-heavy but rigid solution, the project emphasizes modularity, scalability, and a low barrier to entry. The system is intended to function effectively as a Capstone I deliverable while also laying a solid foundation for future academic work and potential product evolution beyond the scope of the course.

2.4.2. Specific Objectives

To support the main objective, the project defines the following specific objectives:

- **Provide a low entry point for digital adoption**: Modula is designed to allow small businesses to adopt a POS system without requiring immediate investment in dedicated hardware or complex infrastructure. By supporting web-based operation on mobile devices and shared terminals, the system enables businesses to experiment with digital workflows before committing to additional costs.
- **Support gradual scaling aligned with business growth**: The system is structured to accommodate both single-outlet businesses and multi-branch organizations. Modula allows businesses to start small and scale up by adding branches, users, and features as operational needs evolve, without requiring architectural redesign.
- **Apply modular system design principles:** A key technical objective is to apply modular design at both the architectural and implementation levels. Core system

responsibilities and feature-specific logic are clearly separated, enabling individual modules to evolve independently while maintaining overall system coherence.

- **Ensure operational reliability and data integrity**: As a POS system handles financial transactions and inventory data, Modula prioritizes correctness and consistency. The project aims to demonstrate reliable handling of sales, orders, inventory updates, cash sessions, and attendance records, even in the presence of connectivity constraints.
- **Support real-world operational workflows**: Modula is designed around realistic business workflows, such as order processing, cash handling, staff attendance, and reporting. These workflows are modeled to reflect how small businesses operate in practice, rather than idealized or purely theoretical processes.
- **Enable offline-aware operation**: Recognizing that network reliability cannot be assumed in all operating environments, the system incorporates offline-aware design principles. While full offline synchronization is deferred beyond Capstone I, the architecture is designed to support queued actions and controlled synchronization in future iterations.
- **Demonstrate alignment between documentation, design, and implementation**: From an academic perspective, an important objective is to maintain consistency between requirement analysis, modular specifications, architectural design, and actual implementation. The project intentionally emphasizes clarity of documentation and iterative refinement to reflect realistic software development practices.

Together, these objectives define Modula as both an academic artifact and a practical system prototype. The project does not aim to deliver a fully production-ready POS platform within Capstone I; instead, it focuses on demonstrating sound architectural decisions, realistic operational coverage, and a clear path for future development. By aligning technical execution with real-world business needs, Modula seeks to validate the feasibility and relevance of a modular POS system developed within an academic context.

2.5. Project Planning

The planning of Modula POS was organized as a phased development plan over a total duration of twelve weeks. While the project was initially scoped as a capstone prototype, the team's implementation trajectory moved toward a high-fidelity prototype due to the level of functional integration achieved across modules. As a result, project planning emphasized both

structured phase progression and iterative refinement as new technical and business requirements were discovered during implementation.

2.5.1. Planning Structure and Timeline

Figure 1 presents the high-level timeline of the project phases over the twelve-week duration, emphasizing how initiation, system design, UI/UX design, and development activities were scheduled to overlap. This structure supports concurrent frontend and backend development while allowing iterative refinement of module specifications as implementation progressed.

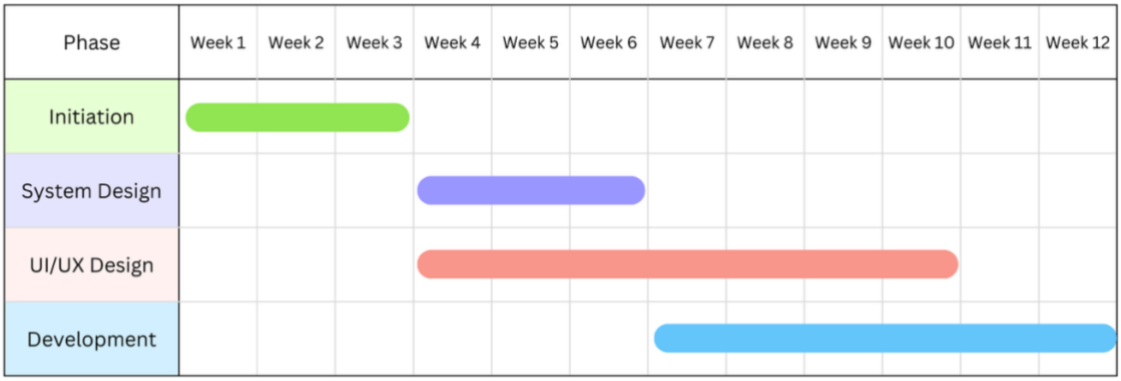

Figure 1: Project Timeline

The project was planned using a phase-based structure consisting of the following stages:

- **Initiation (Week 1 - 3):** Proposal preparation, initial requirement gathering, and background study to build a shared understanding of Point of Sale systems and their operational needs.
- **System Design (Week 4 - 6):** Definition of the overall architecture, early module specifications (formal descriptions of module responsibilities, boundaries, and interactions), and initial separation between core system responsibilities and feature modules.
- **UI/UX Design (Week 4 - 10):** Production of detailed wireframes covering key functional areas, including login, portals, policy configuration, inventory, menu management, sales, cash sessions, order handling, reporting, and receipt views.
- **Development (Week 7 - 12):** Frontend and backend implementation based on the UI/UX designs and evolving module specifications.

In practice, these phases were not executed strictly sequentially. Instead, they overlapped intentionally to support parallel development. For example, once UI/UX designs for early workflows were completed (such as login), backend development could begin on related modules (such as authentication and tenant context), while frontend developers implemented the corresponding UI and view-model layers using hard-coded data or mock repositories. Once backend APIs became available, frontend integration replaced mocks with real data sources using backend-provided API contracts.

This overlapping development model improved productivity under limited team resources and reduced idle time caused by sequential dependencies.

2.5.2. Team Organization and Responsibilities

The project was implemented by a six-member team. The responsibilities of each member are outlined in the table below.

Team Member	Responsibilities
Lim Ieangzorng	Lead Development, Documentation, Acceptance Testing and Frontend Development
Tong Vorak	Frontend Development
Rith Seyhak	Backend Development
Chey Rotana	Backend Development
Chhun Nika	UX/UI Designer
Kon Sotheara	UX/UI Designer

Table 1: Team Members and Responsibilities

This structure supported specialization while allowing the lead developer to coordinate integration efforts and maintain consistency across documentation and implementation.

2.5.3. Work Coordination and Communication

Work coordination followed a task-driven approach rather than daily meetings. Because all team members were full-time students, the project did not adopt daily stand-ups. Instead, coordination occurred through:

- Kickoff meetings at the start of assigned tasks to clarify requirements and expected outcomes.
- Evaluation meetings upon task completion to review implementation results and validate alignment with module specifications.

Meetings were typically conducted one-to-one between the lead developer (Z) and the team member responsible for the task, enabling focused technical discussion and faster decision-making.

Progress tracking was managed using a lightweight Scrum-style board during development, with the intention to transition toward a more formal tool such as Jira for improved visibility and reporting.

2.5.3. Planning Adaptation and Specification Refinement

A defining characteristic of the project planning process was the continuous refinement of module specifications as implementation progressed. Early planning identified functional areas such as sales, inventory, reporting, multi-branch support, and staff attendance; however, several components were initially underspecified due to limited team experience in modular system design.

During execution, the team improved planning quality by systematically refining module specifications to:

- explicitly define module boundaries and responsibilities,
- clarify dependencies and cross-module impacts,
- remove ambiguous or impractical business logic, and
- align documentation more closely with what could be implemented within Capstone I constraints.

This refinement led to important discoveries and adjustments. For example, multi-branch support was reinterpreted not as a standalone module but as part of the tenant-and-

branch context. Additionally, the need for a dedicated cash session module emerged after identifying integrity risks in sales workflows.

The UI/UX development process also influenced specification refinement. While wireframes provided strong structural guidance, interaction details were often faster to validate through frontend prototypes than through exhaustive design-tool simulation. As a result, manual testing and real usage flows became an important feedback mechanism that informed iterative improvement of both the module specifications and system behavior.

Overall, the project planning approach can be characterized as goal-driven and iterative: a structured phase plan provided direction, while continuous refinement ensured that the system remained feasible, consistent, and aligned with operational constraints discovered through development.

III. LITERATURE REVIEW

3.1. Foundation of POS System and Modular Software

3.1.1. Introduction to POS

A Point of Sale (POS) refers to the location or moment where a sale occurs, marking the transfer of goods from the retailer to the customer and accounting for applicable taxes. A POS system extends this concept by providing software-driven management of sales operations, offering real-time control over inventory, sales analysis, and near-real-time insights to support informed decision-making [1].

Historically, POS systems were limited to fixed cash registers and manual inventory tracking. While sufficient for simple transactions, these systems lacked flexibility and real-time data processing. Technological advancements have transformed POS solutions into mobile and cloud-based systems, offering greater speed, mobility, and centralized access to data. Modern POS integrates wireless connectivity (Wi-Fi, Bluetooth), cloud computing, and advanced applications such as sales tracking, inventory management, and analytics, enabling businesses to operate more efficiently [2].

In business, particularly in retail, restaurants, and cafés, POS systems streamline transactions, improve operational efficiency, and support data-driven decision-making. In the

Food and Beverage (F&B) industry, they accelerate order processing, optimize food production, reduce service and payment times, and enhance overall operational control, which is crucial for meeting customer demands and managing high-volume operations effectively [3].

3.1.2. Features of Modern POS System

Modern POS systems are designed to support business operations efficiently by integrating essential modules that handle sales, inventory, staff management, and reporting. These core modules include:

- **Inventory Tracking:** Monitors stock levels in real-time, provides alerts for low inventory, and helps manage restocking and supplier orders.
- **Sales & Transaction Management:** Facilitates smooth and accurate sales processing, supporting multiple payment methods such as cash, credit/debit cards, and digital wallets.
- **Employee Management:** Tracks staff attendance, schedules, and activity reports, helping managers oversee workforce efficiency.
- **Reporting & Analytics:** Generates daily, weekly, and monthly sales reports, identifies top-selling products, and supports data-driven decision-making. [4, 5, 6, 7]

These modules are considered baseline functionalities and are commonly present in most modern POS systems, forming the foundation upon which more advanced or specialized features may be built.

3.1.3 Analysis of Existing POS Systems

❖ **Square POS System (International POS System)**

Square POS is a widely used point-of-sale system designed for small and medium-sized businesses, particularly in retail and café environments. Its core features include:

- **Payment Processing:** Square POS supports multiple payment methods, including card-based, contactless, and digital payments, allowing transactions to be processed through a unified and secure interface.
- **Order Management & Cashier Interface:** The system provides a clean and intuitive user interface for cashiers, enabling staff to place, track, and fulfill orders directly from the POS screen, which helps improve operational efficiency and reduce human error.

- **Sales Reporting & Analytics:** Square offers a centralized dashboard that provides real-time sales data, transaction history, and performance reports, supporting informed business and financial decision-making.
- **Inventory Management:** The platform includes basic inventory tracking tools that monitor stock levels and help businesses manage product availability alongside sales operations.
- **Staff Management:** Square POS incorporates workforce management features such as employee onboarding, time tracking, scheduling, and basic staff administration within the same system. [4]

Despite its comprehensive features, Square POS has some limitations. Its all-in-one design offers limited customization, restricting flexibility for businesses with complex workflows or unique service models. Additionally, as a cloud-based platform, it relies heavily on stable internet connectivity, so operations such as real-time reporting and data synchronization may be disrupted in areas with unreliable or limited internet access, which can affect overall system reliability. [8]

❖ **Toast POS System (International POS System)**

Toast POS is a cloud-based point-of-sale system designed for foodservice businesses of various sizes, ranging from independent cafés and food trucks to full-service restaurants and multi-location chains. Its core features include:

- **Sales Reporting & Analytics:** Offers real-time dashboards with sales summaries, inventory tracking, and performance reports via sale dashboard. It is accessible on the Toast mobile application, enabling managers to monitor performance and make informed decisions in real time.
- **Inventory Management:** Monitors stock levels, supports menu tracking, and helps manage product availability.
- **Offline Mode:** Allows businesses to continue processing transactions even when the internet connection is temporarily unavailable, which is particularly useful in areas with unstable connectivity
- **Payment Processing:** Supports multiple payment methods, including card, contactless, and digital payments, ensuring secure and fast transactions.

- **Order Management & Cashier Interface:** Provides an intuitive interface for staff to take, track, and fulfill orders efficiently, reducing errors and improving service speed. [5, 9]

Despite its functionality, Toast POS setup can be time-consuming even with guided support, and additional advanced features or add-ons may require extra subscriptions, which can increase costs for smaller businesses. [10]

❖ **Champei POS System (Local POS System)**

Champei POS is a Cambodian point-of-sale solution that can be customized to fit every type of store or business. For cafés, its core features include:

- **Multi-Branch Management:** Allows businesses to manage multiple outlets seamlessly, including copying items between branches and viewing aggregated sales summaries for all locations in one place.
- **Sales Reporting System:** Generates daily sales summaries that can be printed, while real-time sales data can be accessed via mobile devices, enabling informed decision-making.
- **Employee Permission System:** Supports adding employees, assigning roles, and defining duties, helping businesses track staff responsibilities and attendance effectively.
- **KHQR Payment Integration:** Natively supports Cambodia's KHQR standard, enabling quick and secure QR-based payments. [11]

Despite its core features, Champei POS does not mention inventory management on its official website, indicating the absence of this key functionality. This limits the system's overall feature richness, though its unique strength lies in the native integration with Cambodia's KHQR payment standard, enabling quick and secure digital transactions.

❖ **MPOS System (Local POS System)**

MPOS is an all-in-one Cambodian POS system tailored for restaurants, cafés, and foodservice businesses. It is designed to streamline daily operations by integrating payment, order, and inventory management into a single platform. The core features of MPOS include:

- **Cashier Management:** Supports multiple payment methods, both online and offline, and allows dual-currency transactions, providing flexibility for diverse customer payment needs.
- **Order Management:** Handles dine-in and takeaway orders, with real-time tracking of order status to ensure timely service and reduce errors.
- **Inventory Management:** Supports procurement, inbound and outbound stock tracking, and real-time inventory monitoring. Dish recipes can be linked to inventory to automatically reduce stock upon sales, providing accurate cost control and early warning for low-stock items. [12]

3.1.4. Discussion of POS Features and Identified Gaps

Based on the core features of modern POS systems outlined in Section 3.2.2, the reviewed international and local POS systems demonstrate varying levels of feature support and system capability.

International POS systems, including Square and Toast POS, generally support most core functionalities such as transaction management, inventory tracking, employee management, and advanced reporting. These systems offer robust analytics, scalable cloud-based infrastructures, and mature system stability. In addition, both Square and Toast provide offline operation capabilities, allowing transactions to be processed temporarily without an active internet connection and synchronized once connectivity is restored. [4, 5] However, despite these strengths, international POS solutions often lack flexibility in adapting to localized business workflows and do not natively support local digital payment standards. Customization for regional requirements may require additional configuration or third-party integrations.

Local POS systems are primarily designed to address regional business needs. They commonly support localized features such as KHQR integration, dual-currency transactions, and simplified operational workflows suitable for small and medium-sized enterprises. Despite these advantages, some local systems offer limited inventory management, basic reporting capabilities, and reduced scalability when compared to international solutions.

The analysis indicates that no single existing POS system fully satisfies all core POS features while simultaneously providing architectural flexibility and strong localization support. These limitations highlight the need for a POS solution that integrates comprehensive core functionality with adaptability to local business environments.

In response to these findings, the proposed **Modula POS system** is designed to support all identified core features including sales processing, inventory management, employee management, and reporting while adopting a modular architecture to enhance flexibility, scalability, and ease of future expansion.

3.2. Modular Software Concept

Modular software is a design approach in which a system is built from independent and interchangeable modules, each responsible for a specific function. These modules communicate through clearly defined interfaces, enabling changes or extensions to be made without affecting the entire system. [13, 14]

The key advantages of modular software include flexibility, scalability, and ease of maintenance. Modular architecture allows features to be introduced gradually through phased implementation, while updates and fixes can be applied to individual modules instead of the whole system. This reduces system complexity and supports long-term adaptability to evolving requirements. [13, 15]

At a high level, modularity in software is commonly illustrated by business applications that separate major functionalities into distinct components. For example, systems may organize user management, reporting, and data handling as independent modules that can evolve separately. This example is used to illustrate modular design principles rather than describe a specific implementation. [14]

In the context of Point of Sale (POS) systems, modular software architecture is particularly relevant due to the diverse and evolving operational needs of businesses. A POS system typically encompasses multiple functional areas, such as sales processing, inventory management, employee management, reporting, and payment integration. By adopting a modular architecture, these functionalities can be developed, deployed, and maintained as independent modules, allowing POS systems to adapt more easily to business growth, regulatory changes, and localized requirements. This architectural approach enables POS solutions to support scalability, customization, and long-term maintainability, making modular design a suitable foundation for modern POS systems that must balance core functionality with flexibility and extensibility.

IV. PROJECT ANALYSIS AND CONCEPTS

This section presents a comprehensive analysis of the Modula system, focusing on its functional and non-functional requirements, system users, and core operational workflows. The purpose of this section is to translate the project's vision and architectural decisions into clearly defined system behaviors and quality attributes, ensuring that the proposed solution is both feasible and aligned with real-world operational needs of Food & Beverage (F&B) businesses.

The analysis in this section is grounded in the finalized module specifications and the implemented business logic developed during Capstone I. Throughout the project, multiple iterations were performed to refine system boundaries, clarify responsibilities between core and feature modules, and resolve ambiguities discovered during implementation. As a result, the requirements and behaviors discussed in this section reflect practical design choices rather than hypothetical or purely conceptual features.

To maintain academic rigor and scope control, this analysis focuses exclusively on the functionalities delivered within the Capstone I phase. Advanced features such as automated subscription billing, cross-branch analytical aggregation, hardware-level POS integrations, and notification systems are intentionally excluded and reserved for future development phases. This ensures that the presented requirements remain realistic, testable, and achievable within the defined project timeline.

The section is structured as follows. Section 4.1 defines the functional requirements of the system, describing what the system must do to support daily operations such as sales processing, inventory management, staff attendance, and cash handling. Section 4.2 outlines the non-functional requirements, addressing system qualities including security, performance, usability, reliability, portability, and maintainability. Finally, Section 4.3 analyzes the system from a user and behavioral perspective by identifying system actors and illustrating their interactions through use case and activity diagrams.

Together, these analyses provide a clear and structured understanding of how Modula operates as a modular, scalable POS system and how its design decisions support both current operational needs and future system evolution.

4.1. Functional Requirements

This subsection defines the functional requirements of the Modula system, specifying the core behaviors the system must provide to support daily operations of Food & Beverage

(F&B) businesses. The requirements are derived from finalized module specifications, validated business logic, and practical implementation during Capstone I. Each requirement is written to be clear, testable, and aligned with the approved project scope.

The functional requirements are organized by operational domain for clarity and traceability.

No.	Module	Functions
1	User Authentication and Authorization	• The system shall allow users to authenticate using a phone number and password. • The system shall support role-based access control with predefined roles, including Admin, Manager, and Cashier. • The system shall restrict access to features and data based on the authenticated user's role and assigned branch context. • The system shall allow a single user account to be associated with multiple tenants, enabling one user to manage multiple businesses. • The system shall maintain authenticated sessions to ensure continuity during active operations.

2	Sales and Order Processing	• The system shall allow authorized users (Admin, Manager, Cashier) to create sales by selecting menu items and applicable modifiers. • The system shall support different sale types, including dine-in, take-away, and delivery. • The system shall allow users to review and modify a cart before checkout, including item quantities and selected modifiers. • The system shall calculate item-level subtotals, item-level discounts, branch-level discounts, value-added tax (VAT), and grand totals. • The system shall support transactions in both USD and Khmer Riel (KHR), displaying transaction amounts in both currencies. • The system shall finalize sales into orders with defined lifecycle states such as in-prep, ready, delivered, void-pending, and voided. • The system shall allow cashiers to request voiding of finalized sales, subject to approval by a Manager or Admin. • The system shall prevent unauthorized modification of finalized or voided sales while preserving a complete audit trail.

3	Inventory Management	• The system shall allow administrators to create, update, archive, and restore stock items. • The system shall allow stock items to exist without a category and be treated as uncategorized. • The system shall maintain inventory quantities at the branch level. • The system shall automatically deduct inventory quantities when a sale is finalized, subject to inventory policy settings. • The system shall record all inventory changes using a journal-based approach to preserve a complete history of stock movements. • The system shall support expiration date tracking for inventory items where enabled.
4	Menu Management	• The system shall allow administrators to create, update, archive, and restore menu items. • The system shall allow menu items to be organized into categories while permitting items without categories to remain uncategorized. • The system shall allow administrators to create and manage modifier groups and assign them to menu items. • The system shall support modifier-based price adjustments that contribute to the final item price. • The system shall allow menu items to be assigned to one or multiple branches and removed from branch availability without deletion. • The system shall preserve historical sales data even when menu items are archived.

5	Cash Session and Cash Handling	• The system shall allow authorized users to start and close cash sessions with recorded opening and closing cash amounts. • The system shall associate all cash-based sales with an active cash session when required by policy. • The system shall support cash movements including paid-in and paid-out transactions, subject to tenant policy settings. • The system shall require managerial or administrative approval for sensitive cash operations such as refunds and over-limit paid-outs. • The system shall calculate expected cash totals and cash variance upon session closure.
6	Staff Attendance Management	• The system shall allow staff members to check in and check out based on branch context. • The system shall enforce attendance policies such as out-of-shift approval and early check-in buffers. • The system shall prevent staff from being checked in at multiple branches simultaneously. • The system shall allow administrators and managers to view attendance records across assigned branches. • The system shall preserve attendance records as immutable historical data.

7	Discount Management	• The system shall allow administrators to create percentage-based discounts. • The system shall support item-level and branch-level discounts with defined stacking behavior. • The system shall apply discounts automatically during sale calculation once configured. • The system shall lock discounts into finalized sales to prevent retroactive modification. • The system shall allow managers and cashiers to view applicable discounts without modification privileges.
8	Reporting and Records	• The system shall generate sales reports summarizing transactions within a selected time period. • The system shall include pending void transactions in reports with appropriate status indicators. • The system shall generate inventory reports reflecting current stock levels. • The system shall restrict reporting access to authorized administrative roles.
9	Offline Operation and Synchronization	• The system shall allow sales, inventory updates, and attendance actions to be recorded offline. • The system shall queue offline actions locally and synchronize them when connectivity is restored. • The system shall ensure idempotent synchronization to prevent data duplication or loss. • The system shall resolve conflicts using server-side validation and policy enforcement.

Table 2: Functional Requirements of the Modula POS System

Table 2 summarizes the functional requirements of the Modula system, organized by module to define the core functionalities supporting daily F&B operations. Each requirement is written to be clear, testable, and aligned with the approved project scope and validated business logic.

4.2. Non-Functional Requirements

This subsection defines the non-functional requirements of the Modula system, focusing on the quality attributes that determine how well the system performs its intended functions. Unlike functional requirements, which describe what the system does, non-functional requirements describe the conditions under which the system operates and the qualities it must exhibit to be reliable, usable, and maintainable in real-world environments.

The non-functional requirements presented here are derived from the operational context of small and medium-sized F&B businesses in Cambodia, technical constraints identified during development, and best practices observed in mature POS systems. These requirements are scoped strictly to Capstone I and reflect deliberate design trade-offs made during implementation.

No.	Module	Functions
1	Security	• The system shall enforce role-based access control to restrict sensitive actions such as voiding sales, managing inventory, and viewing reports. • The system shall ensure that authentication credentials are securely handled and never exposed in plaintext. • The system shall prevent unauthorized modification of finalized transactional data, including sales, inventory records, and attendance logs. • The system shall record critical actions in an audit log to support traceability and accountability.

2	Performance	• The system shall support rapid item selection and checkout workflows to minimize customer wait times. • The system shall perform core operations such as sale creation and cart calculation with minimal latency. • The system shall continue to function during intermittent network connectivity without blocking essential operations.
3	Usability	• The system shall minimize manual decision-making during sales by automating calculations such as discounts, tax, and currency conversion. • The system shall provide role-specific user interfaces, ensuring that users only see features relevant to their responsibilities. • The system shall present policies and configuration options in a structured and understandable manner, similar to familiar settings interfaces. • The system shall allow common workflows, such as sales and attendance actions, to be completed with minimal steps.
4	Reliability	• The system shall ensure that transactional data remains consistent even when operations are performed offline. • The system shall use journal-based recording for critical data such as inventory movements and cash sessions to prevent data loss. • The system shall ensure that partial or failed operations do not result in inconsistent system states.

		• The system shall preserve historical records even when entities such as menu items or staff accounts are archived.
5	Portability	• The system shall be accessible through modern web browsers on phones, tablets, and computers. • The system shall not require dedicated POS hardware to perform core operations in Capstone I. • The system shall support future extension to mobile applications without fundamental architectural changes.
6	Maintainability	• The system shall adopt a modular architecture separating core modules from feature modules. • The system shall allow individual modules to evolve independently without requiring full system redesign. • The system shall centralize cross-cutting concerns such as policy enforcement and audit logging. • The system shall limit inter-module coupling to clearly defined interfaces.
7	Scalability	• The system shall support multiple tenants within a single deployment. • The system shall support expansion from single-branch to multi-branch operations. • The system shall allow new feature modules to be added without disrupting existing workflows.

Table 3: Non-Functional Requirements of the Modula POS System

Table 3 presents the non-functional requirements of the Modula system, organized by module to define the quality attributes that ensure the system operates reliably, efficiently, and securely in real-world F&B business environments.

4.3. Analysis of the Project

4.3.1. System Users

This subsection identifies the primary users of the Modula system and analyzes their roles, responsibilities, and access boundaries. Clearly defining system users is essential to ensure that functionalities, permissions, and workflows are aligned with real operational practices in Food & Beverage (F&B) businesses.

Modula adopts a role-based user model to balance operational efficiency, security, and accountability. Each user role is associated with a specific set of responsibilities and permissions, enforced by the authentication and authorization mechanism of the system.

❖ **Admin**

The admin represents the business owner or a trusted individual responsible for overall system configuration and oversight.

- **Responsibilities:**
  - Configure tenant-level settings such as policies related to tax, currency, inventory behavior, attendance, and cash handling.
  - Manage core operational data including menu items, inventory items, and branch information.
  - Create and manage staff accounts and assign roles.
  - View system-wide records such as sales reports, inventory reports, attendance logs, and cash session summaries.
  - Approve sensitive operations such as sale void requests, cash refunds, and out-of-shift attendance when required.
- **Permissions:**
  - Full access to all system modules within the tenant context.
  - Read and write access to administrative features.
  - Approval authority for restricted or high-risk actions.

The admin role ensures centralized control and accountability for business-critical decisions.

❖ **Manager**

The Manager role represents supervisory staff responsible for overseeing daily operations at one or more assigned branches.

- **Responsibilities:**
  - Monitor sales, orders, and inventory status within assigned branches.
  - Approve operational requests such as sale voids, cash refunds, or out-of-shift attendance, depending on policy settings.
  - Review attendance records and cash session summaries for assigned branches.
  - Support cashiers in resolving operational issues during shifts.
- **Permissions:**
  - Limited administrative access scoped to assigned branches.
  - Approval privileges for selected actions as defined by policy.
  - Read-only access to most configuration settings.

The Manager role acts as an operational intermediary between Admin and Cashier, providing oversight without full system control.

❖ **Cashier**

The Cashier role represents frontline staff responsible for executing sales and basic operational tasks.

- **Responsibilities:**
  - Perform sales transactions by selecting menu items, modifiers, and sale types.
  - Manage active orders, including updating order statuses.
  - Start and close cash sessions when required by policy.
  - Record attendance through check-in and check-out actions.
  - View personal attendance history and assigned work shifts.
- **Permissions:**
  - Access to sales and order-related functionalities.
  - Restricted access to configuration and reporting features.

- No authority to modify finalized sales or system policies.

The Cashier role is designed to minimize complexity and reduce the risk of operational errors by limiting access to only essential functions.

By clearly separating responsibilities and permissions among Admin, Manager, and Cashier roles, Modula ensures operational clarity, security, and traceability. This role-based model forms the foundation for subsequent analysis of use cases and activity workflows in the following sections.

4.3.2. Use Case Diagram

To provide an overall understanding of how users interact with the Modula POS system, a single **high-level use case diagram** is presented. Rather than enumerating every individual system operation, this diagram focuses on the **primary goals and responsibilities** of users when operating the system.

The diagram abstracts internal system processes and emphasizes the relationship between user roles and core system capabilities. Common interactions, such as authentication, are shared across user roles, while management and approval-related use cases are associated only with users holding higher privileges. This approach reflects Modula's role-based access model without introducing implementation-level complexity.

By presenting a consolidated, high-level view, the use case diagram establishes a conceptual foundation for the system's functional scope. More detailed workflows and operational behavior are subsequently illustrated through activity diagrams and module-level descriptions in later sections.

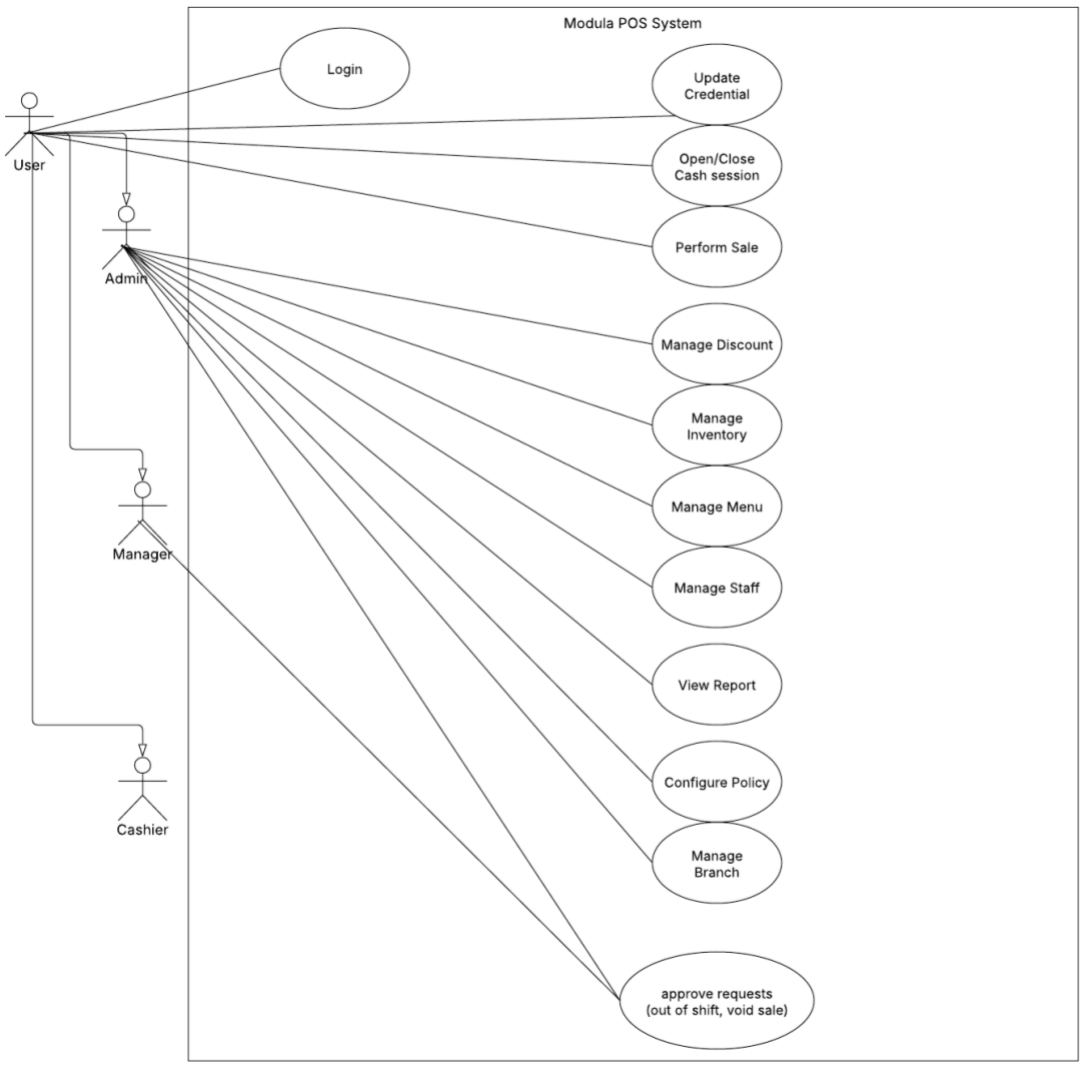

Figure 2: High-level use case diagram of Modula POS System

This diagram presents the primary interactions between system users and the Modula POS system at a conceptual level. It illustrates the main goals that users can achieve such as performing sales, managing menu and inventory, configuring policies, and approving operational requests without exposing internal system processes or implementation details. Role-based access is represented through actor generalization, where different user roles (Administrator, Manager, Cashier) inherit common system interactions while being associated only with the use cases relevant to their responsibilities.

4.3.3. Activity Diagram

Activity diagrams are used in this project to illustrate the dynamic behavior of the Modula POS system from a user-oriented perspective. While the use case diagram presents what interactions are possible between users and the system, activity diagrams focus on how these interactions unfold over time, highlighting decision points, alternative flows, and the coordination between user actions and system responses.

Given the breadth of functionality within Modula, only key workflows were selected for activity modeling. The diagrams included in this section represent critical and representative processes that capture the core operational logic of the system while avoiding unnecessary duplication. Specifically, the following workflows are modeled:

- User Login and Role-Based Navigation, which demonstrates authentication, role resolution, and tenant selection behavior.
- Sale and Order Creation, which represents the primary transactional workflow of the POS system, from item selection to sale finalization.

These workflows were chosen because they:

- are central to daily system usage,
- demonstrate the interaction between users and core system rules.

The activity diagrams intentionally abstract away implementation-level details such as API calls, database operations, session management, and asynchronous processing. Instead, the system is represented as a single logical actor, allowing the diagrams to remain accessible to a non-technical audience while still conveying meaningful system behavior.

Detailed implementation logic corresponding to these workflows is addressed later in the report, particularly in the Implementation chapter.

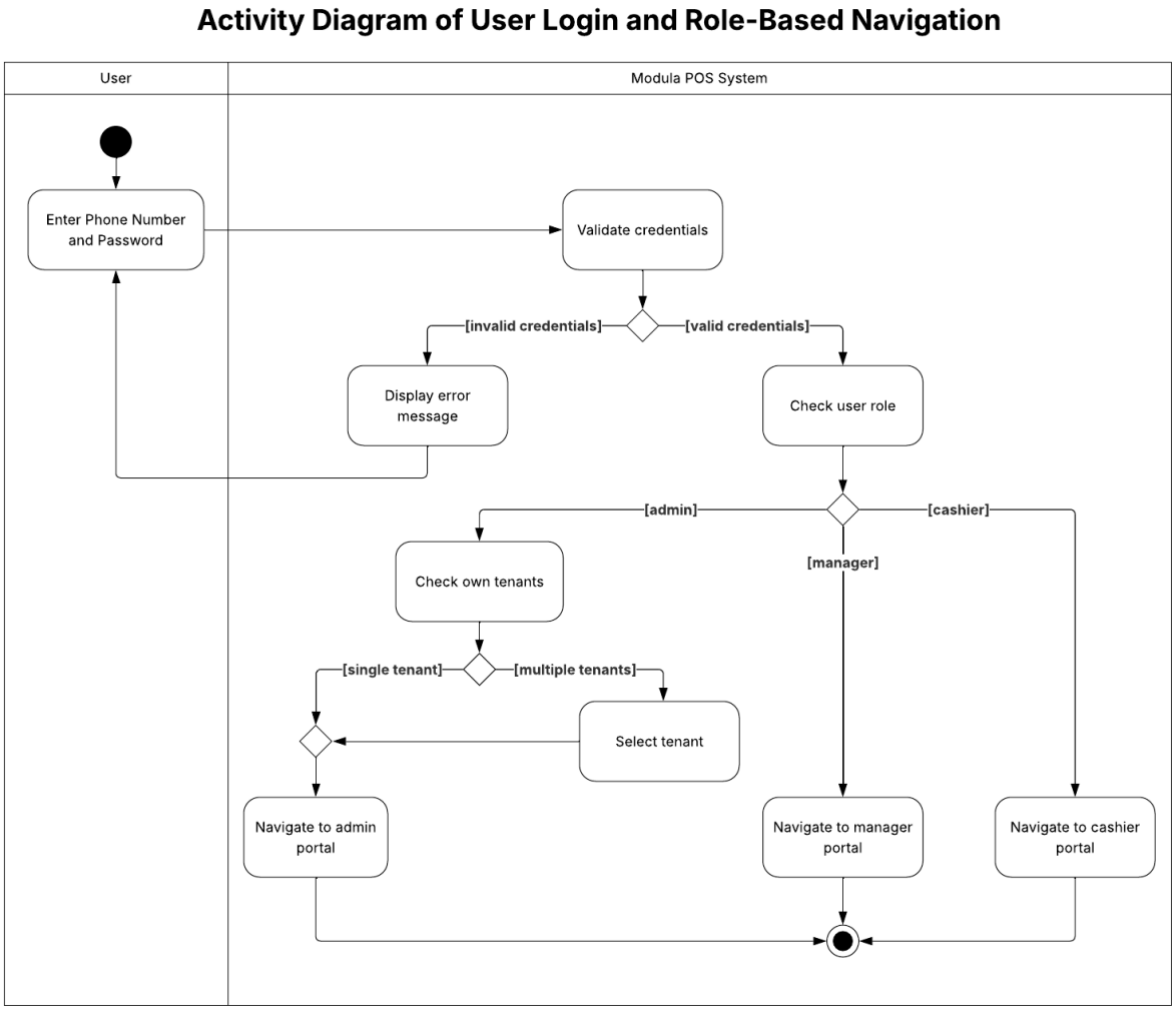

Figure 3: Activity Diagram of User Login

This **Figure 2** illustrates the login workflow of the Modula POS system, showing how user authentication, role verification, and tenant resolution determine system navigation. After entering login credentials, the system validates the user and handles invalid attempts by displaying an error message. Upon successful authentication, the user's role is evaluated. Administrators may be required to select a tenant when multiple tenants are associated with their account before being redirected to the admin portal, while managers and cashiers are routed directly to their respective portals. The diagram abstracts implementation details and focuses on high-level system behavior and decision points relevant to user access control.

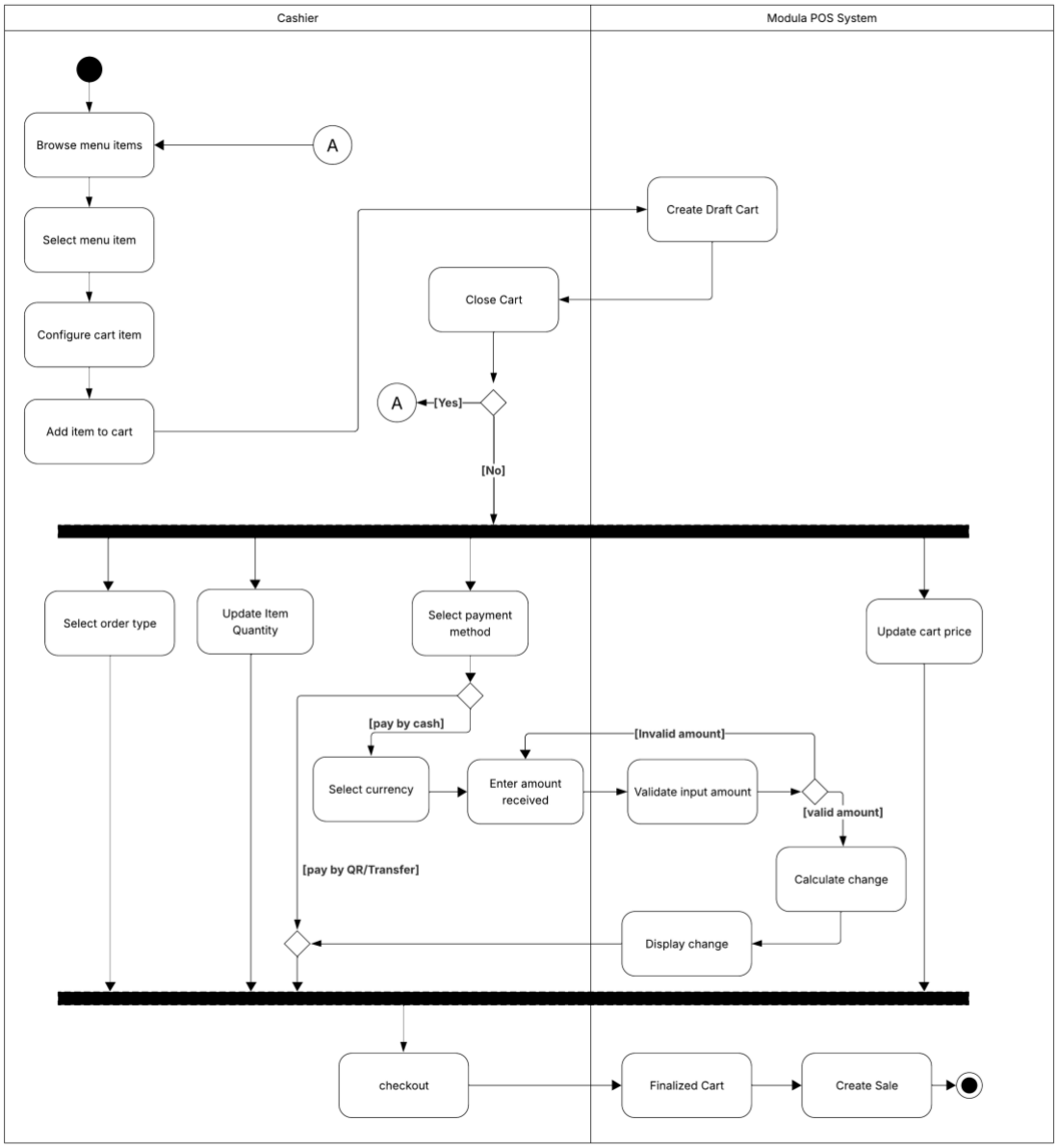

Figure 4: Activity Diagram for Sale and Order Creation

This **Figure 4** illustrates the high-level workflow for creating a sale and its associated order in the Modula POS system from the perspective of a cashier. The diagram abstracts implementation details and focuses on user actions, system responses, and key decision points involved in the transaction process. It highlights the separation of responsibilities between the cashier and the system, the use of a draft cart prior to sale finalization, and the handling of different payment methods. Parallel activities, such as cart configuration and price updates, are shown to reflect real-time system behavior. The diagram also emphasizes that a sale is created only after successful checkout, reinforcing Modula's design principle of separating cart interaction from finalized, immutable transactions.

V. DETAIL CONCEPTS

5.1. Choice of Technology

The Modula POS system was developed using a modern, cross-platform technology stack designed to support scalability, maintainability, and gradual product evolution from an academic prototype into a production-ready system. The selection of programming languages and frameworks was guided not only by technical capability, but also by practical considerations such as team size, long-term maintainability, and alignment with the project's vision of providing an affordable and accessible POS solution for small and medium-sized businesses.

5.1.1. Language and Framework

The Modula POS system was developed using a modern, cross-platform technology stack designed to support scalability, maintainability, and gradual product evolution from an academic prototype into a production-ready system. The selection of programming languages and frameworks was guided not only by technical capability, but also by practical constraints such as limited team size, the need for fast iteration, and Modula's product vision of being device-agnostic with a low entry barrier for small businesses.

In Capstone I, Modula is intentionally delivered as a web-based POS optimized for mobile devices. This design choice aligns with the practical reality of early-stage cafés and small F&B businesses, where operators often start with smartphones or shared devices before committing to dedicated POS hardware. By prioritizing mobile-web usability from the beginning, Modula can be tested and adopted without requiring businesses to purchase printers, drawers, or specialized terminals during the earliest adoption phase. This also supports Modula's long-term goal of scaling "up" (more devices, more staff, more branches) only when the business is ready.

- **Flutter**

Figure 5: Flutter Logo

For the **frontend**, Modula uses **Flutter**, targeting both web and mobile platforms. Flutter was selected primarily for its ability to support cross-platform development using a single codebase. This decision directly aligns with Modula's long-term roadmap: Capstone I delivers a usable web POS experience, while future phases can extend the same system into mobile applications without reimplementing major UI flows or rewriting feature logic from scratch. For a small team, this reduces maintenance overhead and accelerates development because the same UI architecture and component system can be reused across platforms. From an architectural perspective, Flutter supports a modular monolithic application structure, which aligns closely with Modula's system design. Flutter applications are commonly structured using feature-based modules, where each feature encapsulates its own user interface, state management, and interaction logic. This approach improves code organization, readability, and maintainability especially in applications with multiple functional areas such as sales, menu management, inventory, cash sessions, attendance, and reporting. It also reinforces Modula's "build once, configure at runtime" approach, where different roles and tenant capabilities can be represented through runtime-controlled routing and UI visibility.

- **Dart**

Figure 6: Dart Logo

The **Dart programming language** further strengthens this architectural approach through strong static typing, explicit imports, and compile-time error detection. These characteristics allow developers to enforce clear boundaries between frontend modules and reduce the likelihood of runtime errors caused by mismatched data models or unintended dependencies. In Modula, this is particularly important because different user roles—such as administrators, managers, and cashiers interact with different subsets of the system, each with distinct workflows and permissions. Dart's language features also encourage disciplined architectural patterns by supporting immutability, clear data models, and predictable state transitions. These properties are valuable in POS environments, where inconsistent state handling (e.g., duplicated transactions, incorrect order states, or inaccurate totals) can directly cause financial and operational errors.

- **Riverpod**

Figure 7: Riverpot Logo

To manage application state and dependency boundaries within the Flutter frontend, Modula adopts **Riverpod** as its state management and dependency injection solution. Riverpod complements Flutter's feature-based modular structure by allowing application state to be scoped, isolated, and injected explicitly into feature modules. This enables each feature such as sales, inventory, or cash sessions to maintain its own state independently while still integrating cleanly with shared core services (e.g., authentication context, branch context, or policies). The use of Riverpod improves maintainability and testability by reducing tight coupling between UI widgets and business logic. It also supports Modula's modular design philosophy by making dependencies explicit and preventing unintended cross-module interactions. Given the complexity of POS workflows (cart to finalize sale to order state tracking) and the need to handle role-based access and offline-oriented behavior, a predictable and structured state management approach was necessary for Capstone I.

- **Node.js**

Figure 8: Node.js Logo

For the backend, Modula is implemented using TypeScript running on a **Node.js** environment. TypeScript was selected over plain JavaScript to introduce static typing and stronger compile-time guarantees. In a POS system where, multiple modules exchange structured data such as sale transactions, inventory updates, policy enforcement, and cash movement records TypeScript plays a critical role in ensuring consistency and correctness across the system.

- **TypeScript**

Figure 9: TypeScript Logo

TypeScript also supports Modula's modular backend approach by enabling strongly typed contracts and interfaces across modules. This is particularly valuable given Modula's development workflow, where backend modules expose API contracts that the frontend consumes during integration. Strong typing helps reduce payload mismatches, improves developer productivity, and enables safer refactoring as requirements evolve especially important in Modula, where early iterations revealed that overly granular specifications can drift from implementation as business logic becomes clearer through development and testing. From a long-term perspective, TypeScript enhances maintainability and scalability. As Modula grows beyond the scope of Capstone I, additional modules and integrations (e.g., billing automation, hardware integration, OTP services, richer audit and monitoring) can be introduced incrementally. TypeScript helps detect breaking changes early and supports sustainable evolution without destabilizing existing features.

Together, Flutter, Dart, Riverpod, and TypeScript form a complementary technology stack that supports Modula's design philosophy of modularity, clarity, and gradual evolution. Flutter provides cross-platform delivery and supports Modula's device-agnostic direction; Riverpod enforces clean state boundaries in the frontend; and TypeScript ensures reliable modular development on the backend. This combination enables Modula to satisfy Capstone I objectives while remaining aligned with a realistic path toward a scalable product beyond the academic phase.

5.1.2. Databases

The Modula POS system adopts a hybrid data persistence approach that combines a centralized backend database with client-side storage to support both data integrity and operational resilience. Specifically, Modula uses **PostgreSQL** as the primary backend database and **IndexedDB** as a client-side database within the web application. Each database serves a distinct purpose and together they address the practical and technical requirements of a modern Point of Sale system.

- **PostgreSQL**

Figure 10: PostgreSQL Logo

PostgreSQL is used as the authoritative backend database and serves as the system of record for all critical business data. This includes sales transactions, orders, inventory records, cash session data, attendance logs, policies, and audit logs. PostgreSQL was selected primarily for its strong support of **ACID (Atomicity, Consistency, Isolation, Durability)** properties, which are essential in POS environments where financial accuracy and data consistency are critical. Operations such as finalizing a sale, deducting inventory, recording cash movements, and updating reports often involve multiple related data changes that must be committed atomically. PostgreSQL ensures that these operations are executed reliably and consistently, even in the presence of system failures.

In addition, PostgreSQL's relational data model is well-suited to the structured nature of POS data. Modula manages entities such as tenants, branches, users, menu items, stock items, orders, and policies, many of which have well-defined relationships. PostgreSQL allows these relationships to be enforced through foreign keys and constraints, reducing reliance on application-level validation and helping maintain long-term data integrity as the system evolves. Its advanced querying and aggregation capabilities also support reporting requirements such as sales summaries, inventory status, and cash reconciliation, which are fundamental features of POS systems

- **IndexedDB**

Complementing the backend database, Modula utilizes **IndexedDB** as a client-side database within the web application. IndexedDB is used to support offline operation, local caching, and **data synchronization**. In real-world POS deployments, network connectivity may be unreliable or intermittent, particularly in small businesses or mobile environments. IndexedDB enables Modula to continue operating under such conditions by storing operational data locally on the client device.

IndexedDB serves two primary roles in Modula. First, it functions as a temporary persistence layer for actions performed while offline, such as creating sales, updating order

status, or recording attendance. These actions are stored locally and later synchronized with the backend PostgreSQL database once connectivity is restored. Second, IndexedDB acts as a local cache for frequently accessed and relatively stable data, including menu items, categories, modifiers, branch configuration, and policy settings. By caching this data locally, Modula reduces repeated network requests to the backend, lowers data traffic, and improves perceived application performance.

The interaction between **PostgreSQL** and **IndexedDB** follows a clear responsibility separation. PostgreSQL remains the authoritative source of truth for all persistent data, while IndexedDB is used for transient storage and caching on the client side. Synchronization mechanisms ensure that locally stored data is reconciled with the backend in a controlled and consistent manner. This hybrid database design allows Modula to balance reliability, performance, and data consistency, aligning with modern offline-first web application design principles and the operational realities of POS systems.

Overall, the combined use of PostgreSQL and IndexedDB enables Modula to support transactional accuracy, efficient reporting, offline resilience, and reduced network dependency. This database architecture directly supports the system's modular design and contributes to its ability to scale from an academic prototype into a practical, real-world POS solution.

5.1.3. Tools

A set of development, collaboration, testing, documentation, and project management tools was employed throughout the Modula project to support structured implementation, coordination among team members, and consistency between system design and actual behavior. These tools were selected to align with the project's modular architecture and iterative development approach.

- GitHub

Figure 11: GitHub Logo

GitHub was used as the primary version control platform for the project. All frontend and backend source code was managed through GitHub repositories. In addition to code, key project documents such as module specifications, architectural notes, and design decisions were maintained alongside the source code. This approach ensured that documentation evolved together with implementation, improved traceability of changes, and reduced the risk of divergence between design intent and system behavior.

- **Jira**

Figure 12: Jira Logo

Jira was utilized for project management and progress monitoring. Development tasks were organized into issues corresponding to system modules, features, and milestones. Jira provided visibility into task ownership, development status, and outstanding work, supporting an iterative development process and enabling the team to manage scope, prioritize tasks, and respond to design refinements during implementation.

- **Postman**

Figure 13: Postman Logo

Postman was used for API testing and validation during backend development. As Modula follows a modular architecture with clearly defined service boundaries, Postman enabled developers to test backend endpoints independently of the frontend. This facilitated early detection of errors, verification of request and response payloads, and smoother integration between backend and frontend components.

- **Swagger (OpenAPI)**

Figure 14: Swagger (OpenAPI) Logo

Swagger (OpenAPI) was employed to document and standardize backend API contracts. By defining endpoints, request parameters, authentication requirements, and response schemas in a machine-readable format, Swagger provided a clear and consistent interface between backend and frontend development. This reduced ambiguity in data exchange, minimized integration errors, and supported modular development by allowing frontend components to be implemented and integrated based on well-defined API contracts.

- **Visual Studio Code (VS Code)**

Figure 15: Visual Studio Code (VS Code) Logo

Visual Studio Code (VS Code) served as the primary development environment for both frontend and backend implementation. VS Code was chosen for its lightweight nature, extensive extension ecosystem, and strong support for TypeScript, Dart, and Flutter development. Features such as linting, debugging tools, and integrated version control improved development efficiency and code quality.

- **Figma**

Figure 16: Figma Logo

Figma was used for user interface and user experience design. UI layouts, interaction flows, and role-specific interfaces were designed and reviewed using Figma prior to implementation. This enabled early validation of design decisions, improved communication

between designers and developers, and reduced rework during development. Figma played an important role in translating conceptual workflows into implementable interfaces, particularly for complex POS interactions such as sales, cash sessions, and reporting.

Collectively, these tools supported a disciplined and collaborative development process, enabling the team to manage complexity, maintain consistency between design and implementation, and deliver the Modula POS system within the constraints of an academic capstone project.

5.2. Architecture of Web Application

5.2.1. Physical Architecture

The Modula POS system adopts a distributed, web-based physical architecture designed to support accessibility, scalability, and long-term evolution from an academic capstone project into a production-ready software platform. The physical architecture defines how system components are deployed, where they execute, and how they communicate with one another across different environments.

At the time of Capstone I, the physical architecture of Modula is intentionally defined independently of any specific hosting vendor. This decision allows the system to remain provider-agnostic and supports future migration and scaling strategies described in the project's long-term infrastructure vision. Specific hosting providers will be evaluated and selected in Capstone II based on operational, reliability, and cost considerations. By deferring vendor lock-in, Modula maintains architectural flexibility while still establishing a clear and complete deployment model.

- **Client Layer**
    - The client layer consists of end-user devices through which staff and administrators interact with the system. These devices include smartphones, tablets, and desktop or laptop computers commonly used in café and restaurant environments. In Capstone I, Modula is delivered as a web-based application accessed through modern web browsers, while future phases plan to introduce native mobile applications.
    - Client devices are responsible for rendering the user interface, handling user interactions, and managing local application state. To support real-world operating conditions such as unstable or unavailable internet connectivity the

client layer incorporates local data storage using IndexedDB. This enables offline access to essential data such as menus and policies, as well as temporary storage of queued actions that can be synchronized once connectivity is restored. This design directly addresses the operational constraints of small and medium-sized F&B businesses, particularly in regions with inconsistent network reliability.

- **Frontend Deployment**
    - The frontend application is deployed as static web assets generated from the Flutter framework. These assets are served through a web server and may be distributed via a content delivery mechanism to improve load performance and availability. The frontend communicates with backend services exclusively through secure HTTPS-based APIs.
    - From a physical deployment perspective, the public-facing website and the POS application are logically separated, even if they share underlying infrastructure during early deployment stages. This separation reduces operational risk, as traffic spikes or potential denial-of-service events targeting the public website are less likely to impact the POS service used for daily business operations.
- **Backend Application Layer**
    - The backend layer executes as a server-side application that exposes a set of RESTful APIs. It encapsulates the core business logic of Modula, including authentication and authorization, sales processing, inventory management, cash session handling, staff attendance tracking, policy enforcement, and reporting.
    - The backend is designed to be largely stateless, with persistent state stored in the database layer. This approach enables horizontal scaling and simplifies recovery and maintenance. The backend also serves as the central authority for enforcing business rules and access control, ensuring that all operations comply with tenant-specific policies and role-based permissions.
- **Database Layer**
    - Persistent data storage is handled by a relational database management system, specifically PostgreSQL. The database stores critical operational data such as sales transactions, inventory records, staff attendance logs, policy configurations, and audit logs. PostgreSQL was selected for its strong ACID

guarantees, support for complex relational data models, and suitability for financial and transactional systems.

- The database layer enforces data integrity and tenant isolation, ensuring that data belonging to one tenant cannot be accessed or modified by another. Automated backup and recovery mechanisms are part of the intended deployment design, reflecting the system's role as a business-critical application where data loss would have significant consequences.
- Media Storage Service (Images and Assets)
- In addition to structured transactional data, Modula manages media assets primarily images for menu items and stock items. Rather than storing images directly in PostgreSQL, Modula uses an external object storage service for scalability and cost-efficiency.
- In the current implementation, Modula integrates with Cloudflare R2 as the object storage layer. Images are uploaded from the client through backend-controlled APIs, and the backend stores only the necessary metadata (such as object keys/paths and public or signed access URLs, depending on the access model). This approach keeps the database lean and avoids using relational storage for large binary files while still ensuring that assets remain consistently associated with the correct tenant and branch context.
- This external storage component is part of the physical architecture because it is a separate deployed service that the backend depends on at runtime for asset upload and retrieval.
- Identity/OTP Service (Credential Validation Path)
- Modula's authentication design supports secure credential validation and account recovery flows. While Modula currently operates without email-based identity, the intended approach for registration and credential-change verification is OTP (One-Time Password) delivered via SMS (or equivalent provider-supported channel).
- From a physical architecture perspective, OTP delivery is treated as an external service dependency (e.g., an SMS gateway provider) accessed only by the backend. The client requests verification, the backend interacts with the OTP provider; and the backend remains the source of truth for validation and security rules (attempt limits, expiry windows, and abuse prevention). This integration

is considered part of the architecture even if some OTP enforcement steps are still evolving during Capstone I.

- **Local Storage and Offline Support**
    - In addition to server-side persistence, Modula incorporates client-side storage using IndexedDB to support offline-first behavior. When the network is unavailable, the system allows users to continue viewing data and performing permitted actions, which are stored locally and queued for later synchronization. Once connectivity is restored, queued operations are transmitted to the backend in a controlled and idempotent manner.
    - This hybrid storage model improves system reliability and usability, ensuring uninterrupted operations during temporary network outages and reducing unnecessary data traffic by caching frequently accessed reference data.
    - Communication and Integration
    - All communication between system components follows a clear and secure interaction model:
        - Client devices communicate with backend services via HTTPS-based REST APIs.
        - Backend services interact with PostgreSQL through secured database connections.
        - Backend services integrate with external providers (object storage, OTP delivery) using provider APIs and secured credentials.
        - There is no direct communication between client devices and the database or external providers, which helps maintain strong security boundaries and reduces the attack surface.
    - This layered communication model supports centralized enforcement of policies, consistent validation of business rules, and comprehensive audit logging across all system activities.

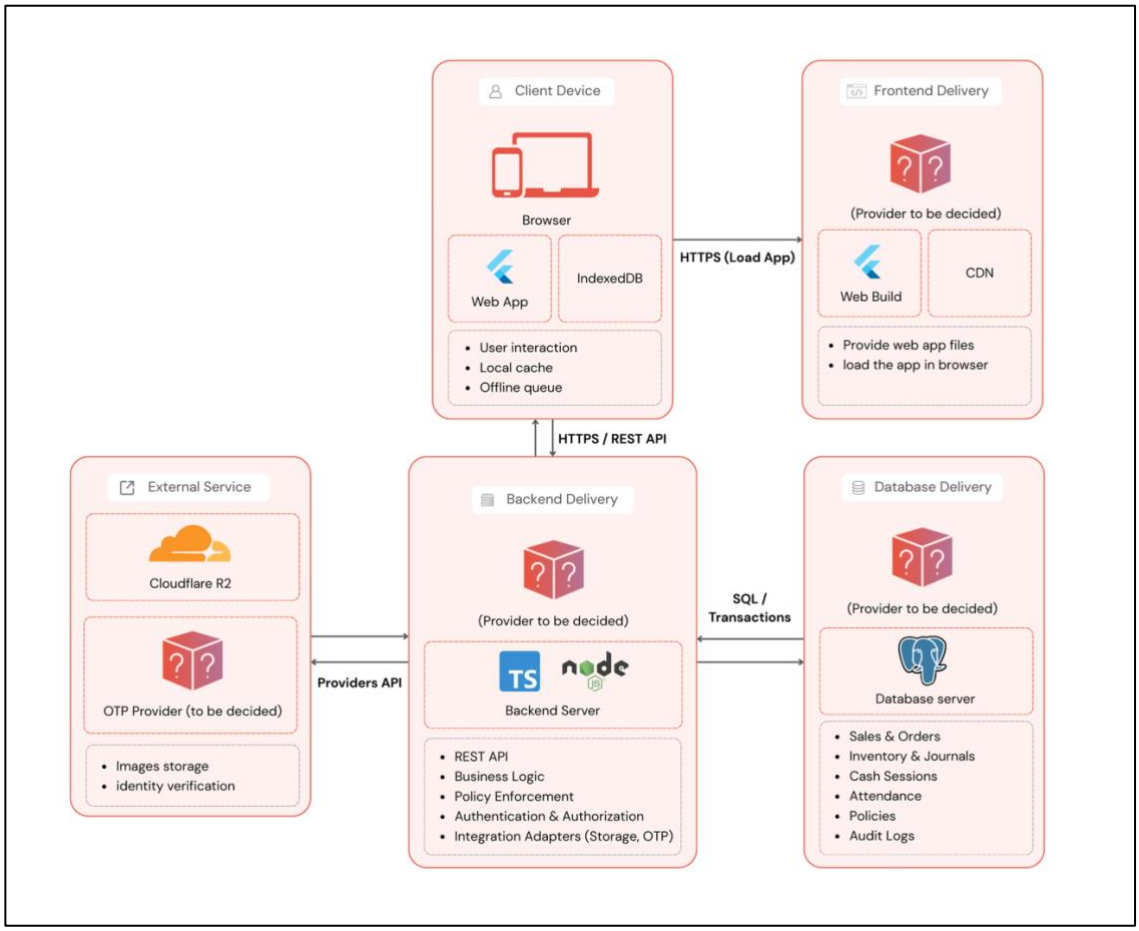

Figure 17: Physical Architecture of Modula POS

5.2.2. Logical Architecture

The Modula POS system adopts a modular monolithic logical architecture, designed to balance clarity, maintainability, and scalability while remaining practical for a small development team and an evolving product roadmap. Rather than decomposing the system into distributed microservices, Modula organizes functionality into clearly defined modules within a single deployable application. This approach reduces operational complexity while still enforcing strong separation of concerns and modular boundaries.

The logical architecture defines how responsibilities are structured within the system, how data flows between components, and how business rules are enforced consistently across different features. By explicitly separating core system responsibilities from feature-specific logic and infrastructure concerns, Modula ensures that the system can evolve incrementally without destabilizing existing functionality.

Layer Architectural Model Modula's logical architecture is organized into four primary layers: the Presentation Layer, the Application (Feature) Layer, the Core Domain Layer, and the Infrastructure Layer. Each layer has a distinct responsibility and communicates with adjacent layers through well-defined interfaces.

- **Presentation Layer**
    - The Presentation Layer is responsible for all user-facing interactions. In Modula, this layer is implemented using Flutter and consists of screens, widgets, and state management logic that render the user interface for administrators, managers, and cashiers.
    - State management within the Presentation Layer is handled using Riverpod, which enables reactive, testable, and dependency-aware state handling. Riverpod supports Modula's modular frontend design by allowing each feature module such as sales, inventory, or attendance to manage its own state independently while still consuming shared application context such as authentication status, tenant identity, branch context, and policy configuration.
    - Importantly, the Presentation Layer does not enforce business rules. Instead, it reacts to capabilities, roles, and policy values provided by the underlying layers. User interface elements may be hidden or disabled based on permissions or policies, but final enforcement occurs outside the UI. This prevents duplication of logic and reduces the risk of inconsistent behavior across different screens or devices.
- **Application (Feature) Layer**
    - The Application Layer contains Modula's feature modules, each responsible for a specific business capability. These include modules such as Sale, Menu, Inventory, Cash Session, Staff Attendance, Discount Management, Reporting, and Receipt handling.
    - Feature modules orchestrate business workflows and coordinate interactions between the Presentation Layer, Core Domain Layer, and Infrastructure Layer. For example, a sale workflow may involve validating policy values, checking cash session status, applying discounts, deducting inventory, generating audit logs, and updating order status. While multiple modules may be involved in this process, the Sale module remains responsible for the overall transaction flow.

- Feature modules do not manage user identity, permissions, or tenant isolation directly. Instead, they consume these concerns from the Core Domain Layer. This separation ensures that cross-cutting rules such as authorization checks or tenant boundaries are applied consistently across all features.

- **Core Domain Layer**
  - The Core Domain Layer contains Modula's core modules, which provide system-wide guarantees and shared services. These include Authentication and Authorization, Tenant and Branch Context, Policy and Configuration, Sync and Offline Support, and Audit Logging.
  - Core modules encapsulate stable domain concepts that should remain consistent even as features evolve. Authentication rules, tenant isolation, and policy evaluation are centralized in the Core Domain Layer rather than duplicated across feature modules. This improves maintainability and reduces the risk of security or data integrity issues.
  - A key architectural principle in Modula is that feature modules depend on core modules, but core modules do not depend on feature modules. This dependency direction reinforces modular boundaries and allows feature modules to evolve without destabilizing core system behavior.
- **Infrastructure Layer**
  - The Infrastructure Layer provides the technical foundations required to execute the system. This includes database access, API communication, offline storage mechanisms, synchronization queues, and idempotency handling.
  - It also contains the integration adapters for external services that are not part of the domain logic but are required operationally, such as:
    - Object storage adapters (e.g., Cloudflare R2 for images)
    - OTP provider adapters (e.g., SMS gateways) for identity verification workflows
  - Infrastructure components are deliberately kept free of business logic. Their role is to enable persistence, communication, and execution not to decide how the system behaves. For example, the Infrastructure Layer uploads an image to object storage, but it does not decide whether the user is authorized to change a menu item; that rule lives in the domain/application layers.

- This separation allows Modula to evolve its infrastructure (such as migrating storage providers or changing deployment models) without rewriting feature or core domain logic.

- **Inter-Layer Communication**
  - Communication between layers follows a strict and predictable flow. The Presentation Layer interacts with the Application Layer through view models and controllers. The Application Layer consults the Core Domain Layer for authorization, policy evaluation, and tenant/branch context, and relies on the Infrastructure Layer for persistence and external integrations.
  - Direct interaction between the Presentation Layer and Infrastructure services is explicitly avoided. This prevents tight coupling between UI code and network/storage logic, improving testability and long-term maintainability.
- **Modularity and System Evolution**
  - By adopting a modular monolithic logical architecture, Modula achieves a balance between flexibility and simplicity. Modules are independently developed and evolved yet remain part of a single cohesive system. This supports Modula's long-term vision of growing from an academic project into a commercial platform, while avoiding premature architectural complexity.
  - The logical architecture also complements Modula's policy-driven design. Policies defined in the Core Domain Layer dynamically influence both feature behavior and user interface rendering, enabling the system to adapt to different business configurations without structural changes.
  - Overall, Modula's logical architecture provides a clear, maintainable, and extensible foundation that aligns technical design with real-world business requirements and long-term system evolution.

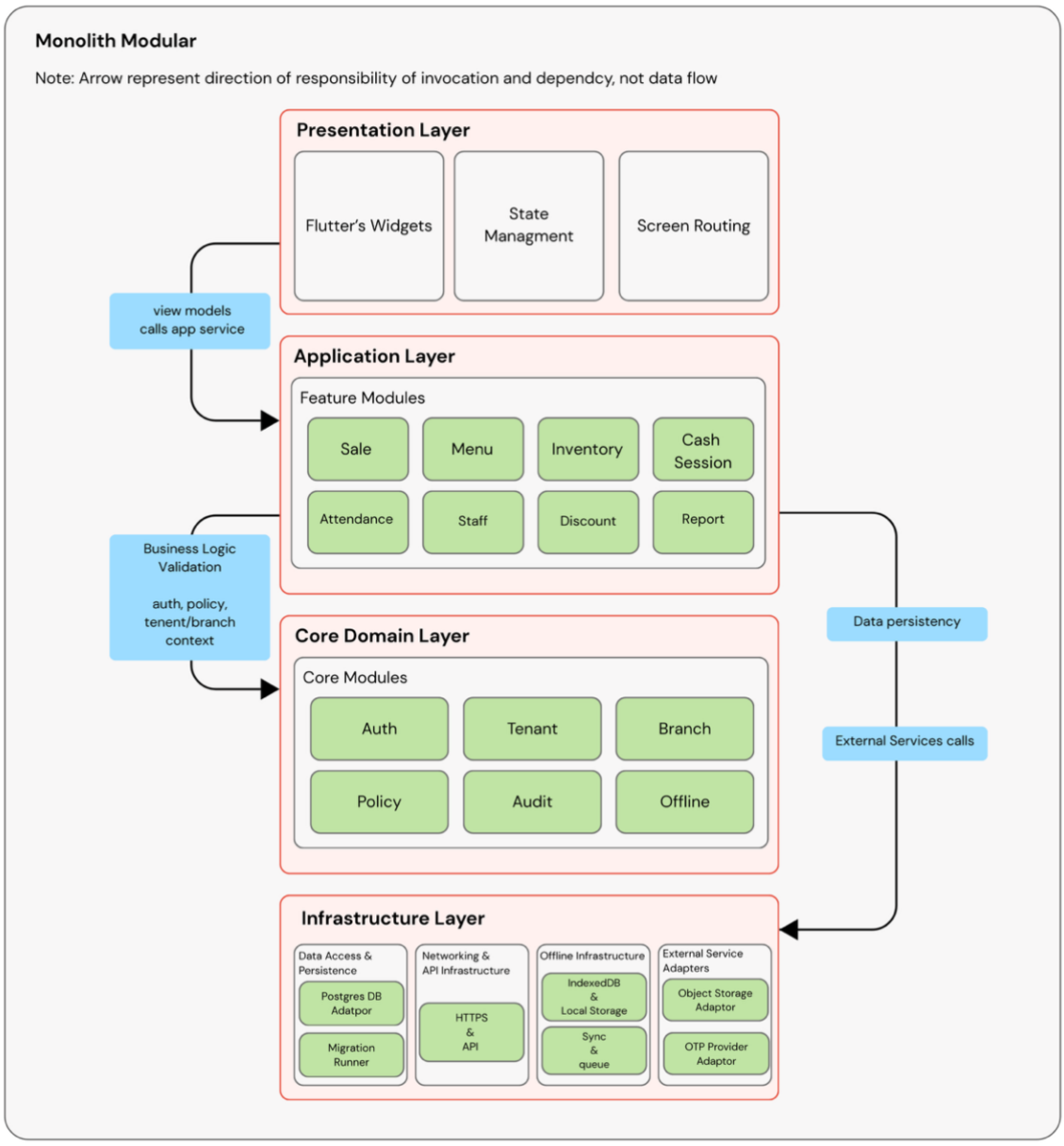

Figure 18: Logical Architecture of Modula POS System

VI. IMPLEMENTATION

This section describes how the Modula POS system was implemented based on the requirements and architectural decisions presented in Sections 4 and 5. Rather than re-stating requirements or design rationale, it focuses on how the proposed modular architecture was realized in practice through the integration of the frontend, backend services, databases, and supporting components such as configuration, offline synchronization, and policy enforcement. The section begins with a system topology overview, followed by project setup and structure,

configuration management, and finally the implementation of core and feature modules delivered in Capstone I.

6.1. Project Topology

This section describes the project topology of the Modula POS system, focusing on how the system's components interact at runtime and how data flows between them during normal operation. While Section 5.2.1 (Physical Architecture) explains where the system is deployed and Section 5.2.2 (Logical Architecture) explains how responsibilities are layered, the project topology provides a runtime interaction view that connects these perspectives. It illustrates how client applications, backend services, databases, and external services cooperate to support core POS workflows.

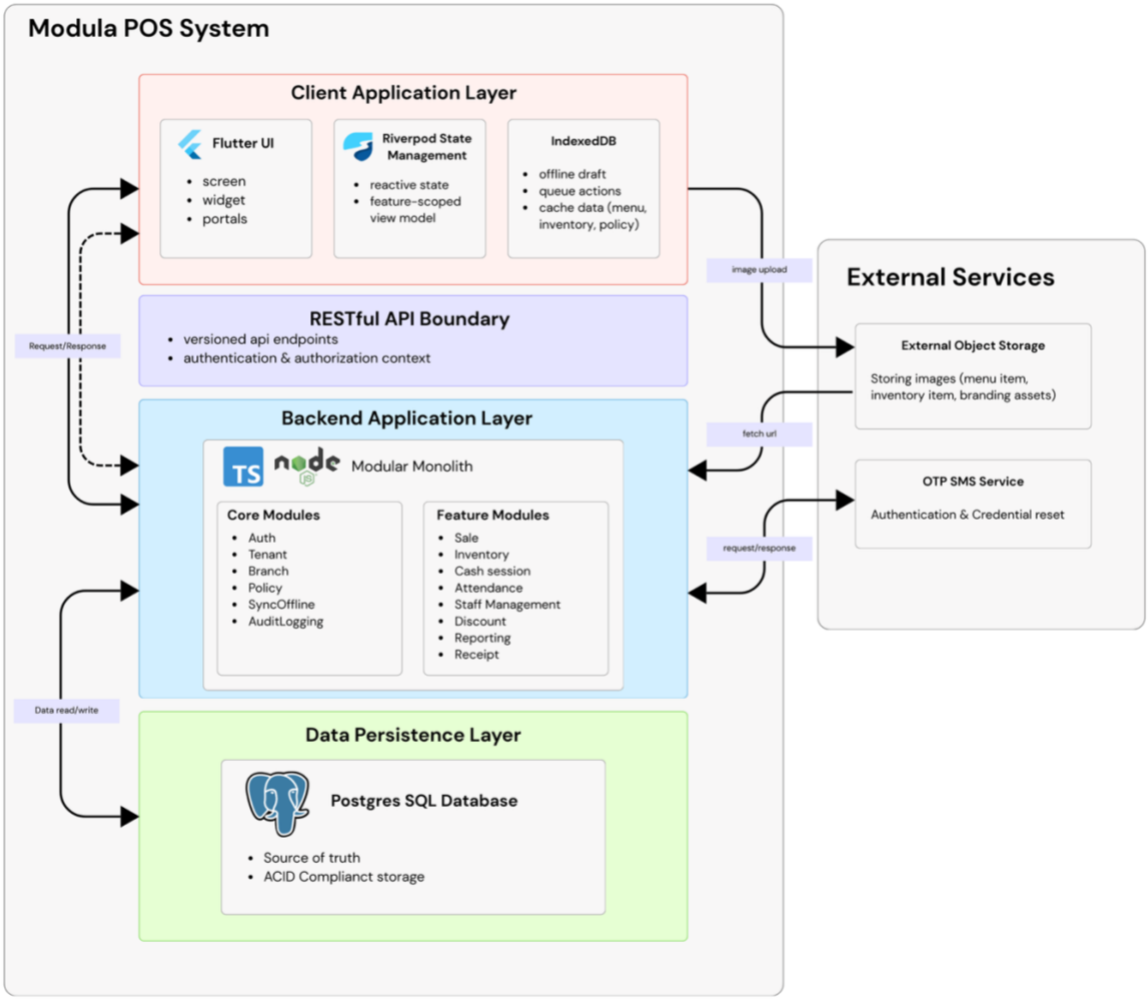

Figure 19: Project Topology

Figure 15 presents a high-level topology of the Modula POS system, illustrating how the client application, backend services, persistent storage, and external integrations interact during system operation. The diagram focuses on component relationships and interaction boundaries rather than internal implementation details.

The topology highlights the separation between the client layer and backend application through a clearly defined API boundary, the use of centralized backend persistence as the system of record, and the inclusion of client-side local storage to support offline-capable workflows. It also shows how the backend integrates with external services, such as object storage and authentication-related services, while maintaining centralized control over business logic and data integrity.

Arrows in the diagram indicate the direction and nature of interactions between components. Solid arrows represent synchronous request–response communication, while dashed arrows represent deferred or asynchronous interactions used for offline synchronization. Together, these interactions demonstrate how Modula balances responsiveness, reliability, and extensibility within a modular system architecture.

The following subsections elaborate on each layer and interaction shown in the topology diagram.

6.1.1. Runtime Components

At runtime, the Modula POS system consists of the following major components:

- **Client Applications:** The client layer is implemented using Flutter and deployed primarily as a web-based POS interface. It is used by Administrators, Managers, and Cashiers to interact with the system. The client is responsible for rendering the user interface, managing local UI state, and supporting offline operations through local storage.
- **Backend API Service:** The backend is implemented as a modular monolithic service using TypeScript. It exposes a set of HTTP APIs grouped by feature modules such as authentication, sales, inventory, cash sessions, and reporting. The backend is responsible for enforcing business rules, applying policies, performing authorization checks, and coordinating data persistence.

- **Primary Database:** PostgreSQL serves as the centralized backend database and acts as the authoritative source of truth for all persistent business data. This includes sales records, inventory data, cash sessions, attendance logs, policies, and audit trails.
- **Client-side Database:** IndexedDB is used within the client application to support offline operation, local caching, and deferred synchronization. It does not serve as an authoritative data source and is only used to temporarily store operational data until it can be synchronized with the backend.External Services:
    - An OTP/SMS service is used during authentication and credential-related workflows.
    - An external object storage service is used for storing media assets such as menu images, inventory item images, and tenant or branch logos.

6.1.2 Interaction Principles

Several principles govern how these components interact:

- Client applications do not directly access the database or external services.
- All business operations and data persistence are mediated by the backend API.
- The backend service is authoritative and enforces consistency, policy rules, and access control.
- Client-side storage supports resilience and performance but never replaces backend validation.
- External services are accessed only by the backend to preserve security and centralized control.

These principles ensure a clear separation of responsibilities and reduce the risk of inconsistent or unauthorized data manipulation.

6.1.3 Meaning of Arrows in the Topology Diagram

In the project topology diagram, arrows represent runtime data exchange

- Arrows from the client to the backend represent synchronous request–response interactions, such as submitting credentials, creating sales, or retrieving menu data.
- Arrows from the backend to the database represent read and write operations on persistent data.
- Dashed arrows between the client and backend represent deferred or asynchronous interactions, such as offline synchronization.

- Arrows between the backend and external services represent service integrations initiated by the backend.

The arrows therefore describe how information flows through the system during execution.

6.1.4 Client–Backend Interaction

The client applications communicate with the backend through a secured API boundary. Typical interactions include authenticating users, retrieving configuration and policy data, creating and finalizing sales, updating order states, and initiating cash session actions. All requests are validated and authorized by the backend before any business logic is executed or data is persisted.

6.1.5 Backend–Database Interaction

The backend interacts bidirectionally with the PostgreSQL database. Read operations retrieve the current system state, such as inventory levels or reporting data, while write operations persist authoritative records, including finalized sales, inventory movements, and audit logs. Transactional guarantees provided by the database ensure consistency across multi-step operations.

6.1.6 Offline Operation and Synchronization

When network connectivity is unavailable, the client records user actions locally using IndexedDB. These actions may include draft sales, attendance check-ins, or order status updates. Once connectivity is restored, the client submits queued actions to the backend. The backend validates each action against current system state and policies before committing it to the database or rejecting it with a clear error.

6.1.7 Integration with External Services

The backend integrates with an OTP/SMS service to support authentication workflows. Requests to send or verify one-time passwords are initiated by the backend, and the responses are used to determine authentication outcomes. Similarly, interactions with external object storage are mediated by the backend, which manages upload processes and stores metadata in the primary database. Clients do not hold permanent credentials for these services.

6.1.8 Summary of Topology

The project topology of Modula emphasizes centralized authority, controlled interaction boundaries, and resilience to unreliable network conditions. By clearly defining how client applications, backend services, persistent storage, and external integrations interact at runtime, the topology supports both the functional requirements of a POS system and the long-term goals of scalability, maintainability, and secure operation.

6.2. Step of Implementation

6.2.1. Project Setup

The implementation of the Modula POS system began with establishing a stable and reproducible project foundation for both frontend and backend development. Given the modular scope of the system and the intention to support parallel development, particular attention was paid to tooling consistency, environment separation, and early architectural alignment.

- **Frontend Project Initialization**
    - The frontend application was initialized using Flutter's standard project scaffolding tool (flutter create), with Flutter version 3.9.2. Web support was enabled from the beginning of the project to ensure that the system could be delivered as a browser-based POS during Capstone I, while remaining compatible with future mobile deployment. Enabling web support early allowed the frontend architecture, routing, and state management to be designed with cross-platform constraints in mind, avoiding later refactoring when extending the system beyond the web environment.
    - The Flutter project structure was prepared to support feature-based modular development, aligning with the modular specifications defined during the design phase. This early alignment ensured that UI components, ViewModels, and data access layers could evolve independently within each feature module.
- **Backend Project Initialization**
    - The backend was initialized using pnpm init, with Node.js version 20.19 and TypeScript configured from the start of development. TypeScript was adopted

immediately to provide static typing, enforce clear module boundaries, and reduce integration errors between backend services and frontend consumers. By avoiding an initial JavaScript-only phase, the project minimized technical debt and ensured that type safety was present throughout the entire implementation process.

  - The backend project was structured as a modular monolithic application, where each domain module could be developed and tested independently while remaining part of a single deployable service. This approach supported incremental development without introducing the operational complexity of microservices at an early stage.

- **Repository Management and Version Control**
  - Frontend and backend codebases were maintained in separate repositories hosted on GitHub. This separation reflected the distinct build processes, dependencies, and deployment considerations of the two applications. A module-based branching strategy was adopted, allowing developers to work on individual modules or features in isolation before merging changes into the main development branch. This approach reduced integration conflicts and supported parallel development across multiple system components.
- **Environment Configuration**
  - Environment-specific configuration was managed using environment variable files. The backend project defined .env.local for local development, .env.development for shared team development, .env.production for future deployment, and .env.example as a reference template. This separation ensured that sensitive credentials were not committed to version control while allowing consistent configuration across different environments. Local development environments were explicitly isolated to prevent accidental coupling with shared or production settings.
- **Database Setup and Migration Strategy**
  - During Capstone I, the system was developed using a local PostgreSQL instance. Rather than relying on an Object-Relational Mapping (ORM) framework, database schema management was handled using raw SQL migration scripts. A custom migration mechanism was implemented to allow all

database migrations to be applied through a single command, simplifying setup for developers and reducing dependency on external tooling.

- This approach provided fine-grained control over database schema evolution and ensured transparency of data structures, which is particularly important for POS systems that require strong consistency and traceability. While Docker-based database deployment was not introduced during Capstone I, the project was designed with containerization in mind for future production environments.

- **Development Tooling**
    - The project standardized on pnpm as the package manager for backend dependency management, ensuring faster installs and consistent dependency resolution. API development and testing were supported using Postman, while Swagger was used to document API contracts and facilitate communication between backend and frontend teams during integration.
- **Initial Architecture Alignment**
    - Before feature implementation began, the team prioritized defining module specifications and system boundaries. Modules were planned first, and implementation details were refined through continuous feedback between frontend and backend development. API versioning using route prefixes such as /v1/auth/ and /v1/menu/ was introduced at the start of development to ensure backward compatibility and allow future evolution of the system without breaking existing clients.
    - This structured setup phase established a solid foundation for subsequent feature development, enabling Modula to be implemented in a disciplined, modular, and scalable manner throughout Capstone I.

6.2.2. Project Structure

This section describes how the Modula POS system is structured at the code level for both frontend and backend implementations. The project structure reflects the design principles established in earlier sections, particularly modularity, separation of concerns, and support for parallel development. Rather than focusing on theoretical architecture, this section explains how those concepts are realized concretely within the source code repositories.

❖ **Frontend Project Structure**

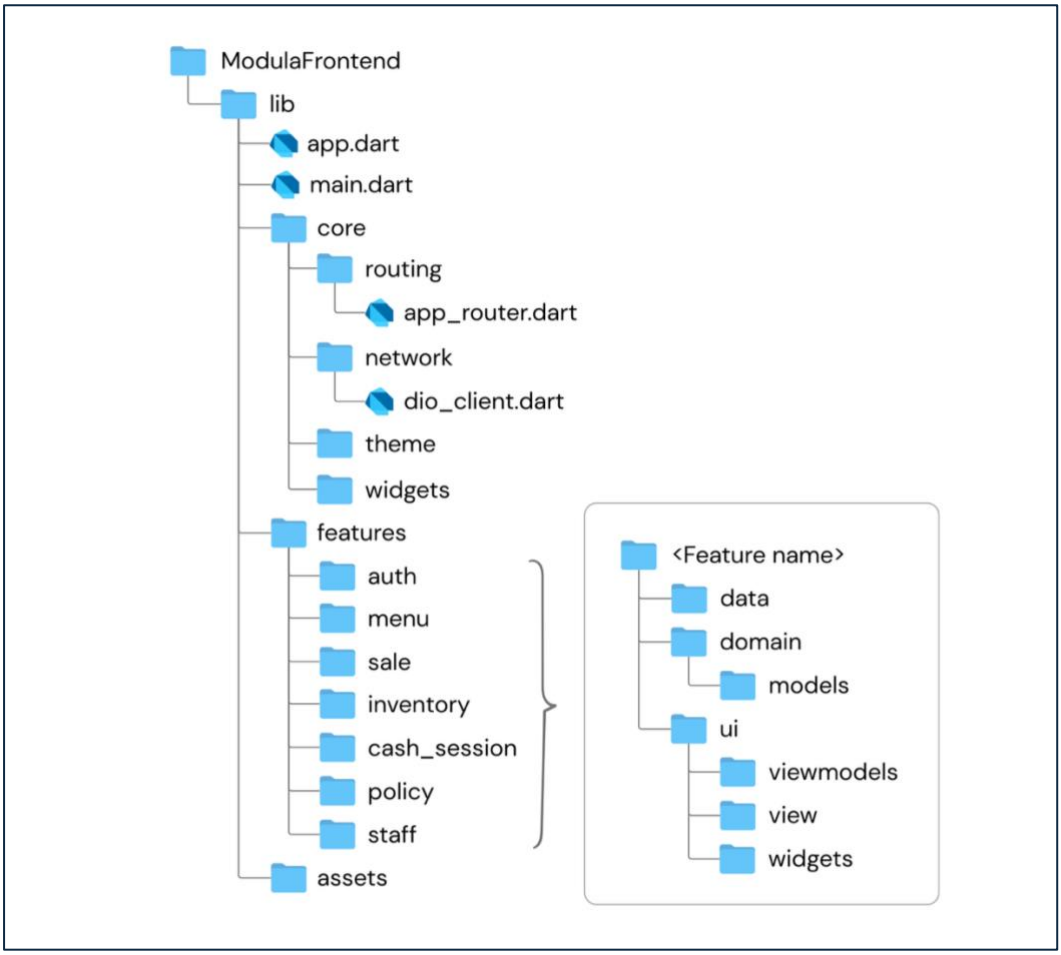

Figure 20: Modula Frontend Project Structure

The frontend of Modula is implemented using Flutter and organized using a feature-oriented modular architecture. Rather than structuring the codebase purely by technical layers, the system is decomposed into functional feature modules that correspond directly to POS capabilities such as sales, inventory, cash sessions, and policy configuration. This structure aligns with the system's modular design principles and supports independent feature development.

Each feature is encapsulated under lib/features/<feature>/ and follows a consistent internal layering pattern. This internal structure separates responsibilities while keeping all logic related to a feature within a single bounded context:

- The data layer handles API communication and repository abstractions.
- The domain layer defines feature-specific models and value objects.

- The UI layer contains screens, widgets, and view models responsible for presentation and interaction.

State management is implemented using Riverpod, with view models encapsulating business interaction logic and exposing reactive state to the UI. Providers are scoped at the feature level, reinforcing modular boundaries and preventing unintended coupling between unrelated features.

At the application level, shared infrastructure such as routing, theming, network configuration, and reusable UI components is centralized under lib/core/. This prevents duplication across features and ensures consistent behavior throughout the application. Navigation is defined declaratively using a centralized router, with role-based access enforced through distinct portals for administrators and cashiers.

To support parallel development, frontend features are initially implemented using mock repositories and placeholder data where backend services are not yet available. Once the corresponding backend module is completed, the mock data layer is replaced with real API integrations. This workflow allows frontend development to proceed independently while remaining aligned with evolving backend contracts.

Overall, this frontend structure balances modularity, maintainability, and development efficiency. It supports Modula's iterative development approach, where UI design, business logic, and backend integration evolve together through continuous feedback and refinement.

❖ **Backend Project Structure**

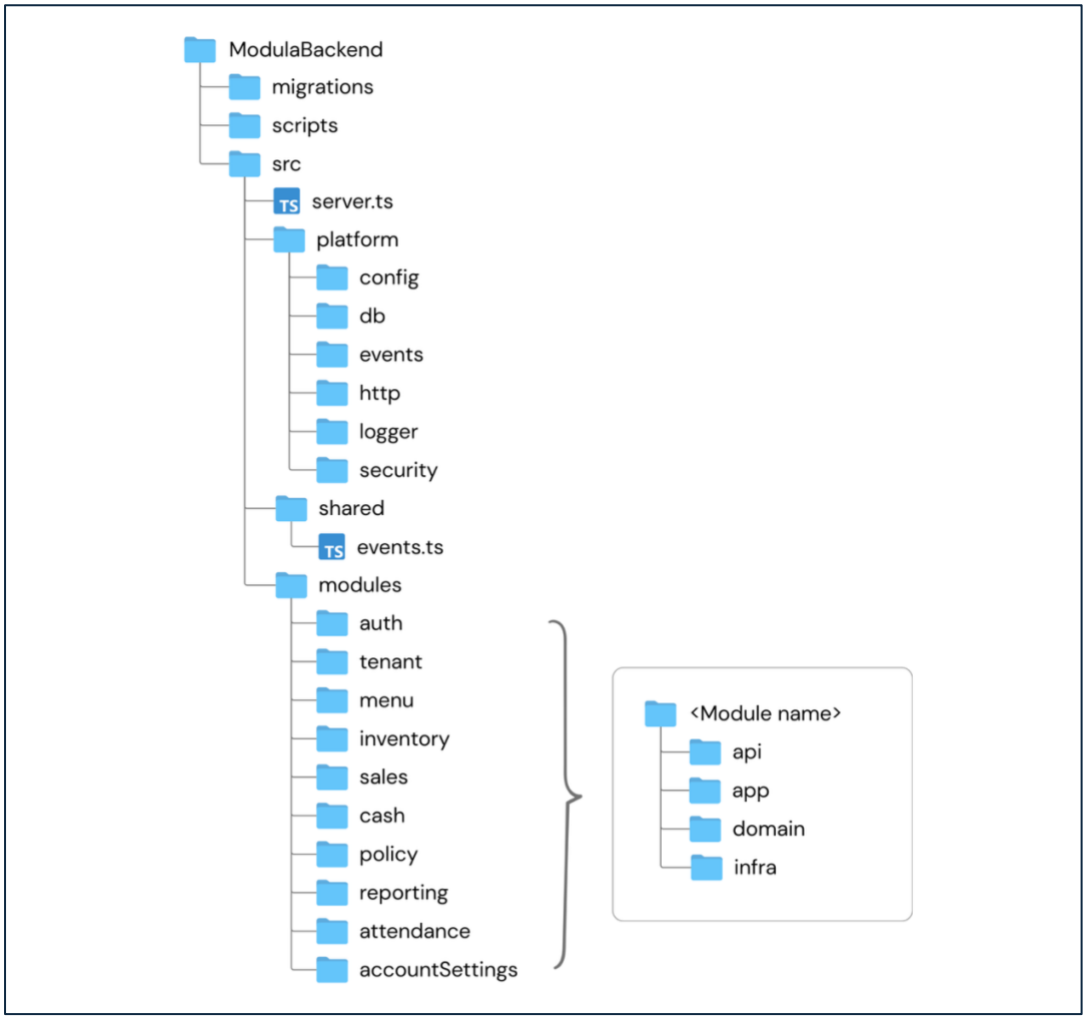

Figure 21: Modula Backend Project Structure

The backend of Modula is implemented as a modular monolithic application using TypeScript and Node.js. Rather than adopting a microservices architecture, the project uses a single deployable service that is internally divided into well-defined modules. This design balances architectural clarity with operational simplicity, which is appropriate for both a capstone project and an early-stage product.

To further clarify how the backend architecture is realized in practice, Figure X presents the actual project directory structure of the Modula backend repository. This textual structure illustrates how the system's conceptual modular architecture is mapped directly into the source code, reinforcing the separation between feature modules, shared infrastructure, and cross-cutting concerns.

The backend follows a clean, layered structure inspired by Clean Architecture principles:

- API layer handles HTTP requests and input validation.
- Application layer contains use cases and application services.
- Domain layer defines core business entities and rules.
- Infrastructure layer implements database access and external integrations.

At the repository level, the backend is organized as follows:

- **src/server.ts** acts as the composition root. It initializes the application, loads configuration, wires module dependencies, registers routes, and starts the HTTP server.
- **src/modules/** contains all feature modules, such as authentication, tenant management, sales, inventory, cash sessions, attendance, policy, and reporting.
- **src/platform/** provides shared infrastructure, including database access, configuration loading, logging, event handling, and security utilities.
- **src/shared/** serves as a shared kernel for common types, errors, and domain event contracts.
- **migrations/** stores raw SQL migration scripts.
- **scripts/** contains developer tooling, such as setup and automation scripts.
- **context/** and **modSpec/** hold architectural documentation and canonical module specifications used during development.

A strict rule is enforced to prevent lateral imports between modules. Modules interact either through defined interfaces (ports) or via domain events. Cross-module communication is handled using an in-process event bus with an outbox pattern, ensuring reliable event dispatch without tight coupling.

Database access is implemented using raw SQL rather than an ORM. Migration scripts are stored as ordered SQL files and executed by a custom migration runner. This approach provides full control over database schema design and avoids abstraction overhead, which is suitable for a POS system requiring precise data handling.

❖ **Alignment Between Frontend and Backend Structures**

The frontend and backend project structures are intentionally aligned around the same set of functional modules. Features such as sales, inventory, cash sessions, and attendance exist on both sides, sharing a common conceptual boundary defined by the module specifications.

This alignment reduces integration friction and supports the project's development workflow. Backend teams implement modules according to the agreed specifications and expose API contracts. Frontend teams consume these contracts when integrating features, replacing mock repositories with real data sources. Versioned API routes (e.g., **/v1/auth**, **/v1/menu**) further support-controlled evolution of the system.

By structuring both codebases around modular features rather than purely technical layers, Modula achieves a high degree of cohesion and clarity. This structure supports parallel development, simplifies maintenance, and enables the system to evolve incrementally beyond the scope of Capstone I.

6.3. Configuration

This section describes how Modula is configured to operate reliably across different environments while remaining secure, portable, and maintainable. Configuration in Modula is designed to externalize environment-specific and sensitive parameters rather than embedding them directly into application code. This approach supports clean separation between implementation logic and operational concerns, aligns with modern software engineering best practices, and enables Modula to evolve from a capstone prototype into a scalable production system.

Rather than binding the system to a specific deployment environment or infrastructure provider, Modula adopts a provider-agnostic configuration strategy. Key system behaviors including database connectivity, API routing, media storage, and runtime settings are controlled through environment variables and standardized configuration modules.

6.3.1. Environment Configuration

Modula uses environment-based configuration to manage differences between development, testing, and future production deployments. Environment variables are used to store sensitive information and environment-specific parameters such as database connection strings, API base URLs, runtime flags, and external service credentials.

The project adopts a layered configuration strategy using the following files:

- **.env.example**: A template documenting all required environment variables without exposing sensitive values.
- **.env.local**: Developer-specific configuration used for local development and excluded from version control.
- **.env.development**: Shared configuration for development environments.
- **.env.production**: Reserved for future production deployment in Capstone II.

This structure ensures that sensitive credentials are never committed to source control while enabling consistent setup across team members. On the backend, environment variables are loaded through a centralized configuration module that performs fail-fast validation at startup. If required variables are missing or misconfigured, the application terminates early, preventing undefined runtime behavior.

On the frontend, environment configuration controls API endpoints and environment flags, allowing the same Flutter codebase to connect to different backend environments without code modification. This supports portability and simplifies future deployment to cloud-based hosting providers.

In addition to application configuration, Modula externalizes credentials for infrastructure services. Product images, menu item images, and inventory-related media are stored using cloud-based object storage rather than within the relational database. At the time of Capstone I, Modula integrates with Cloudflare R2 for object storage. Access credentials, bucket names, and endpoints are provided via environment variables and loaded securely at runtime. This design keeps media storage decoupled from application logic and allows the storage provider to be replaced in the future without major architectural changes.

6.3.2. API Configuration

API configuration in Modula focuses on consistency, versioning, and safe integration between frontend and backend components. All backend APIs are exposed through versioned routes (e.g., **/v1/auth**, **/v1/menu**, **/v1/sale**), enabling future changes and enhancements without breaking existing clients.

The backend includes integrated API documentation using Swagger, allowing developers to explore endpoints, request schemas, and response formats interactively. During development and testing, Postman is used to validate API behavior independently of the frontend, supporting rapid debugging and verification of business logic.

To reduce integration errors and support parallel development, API contracts are treated as first-class artifacts. Backend modules define explicit request and response schemas, which are shared with the frontend team through documented contracts. This contract-driven approach helps prevent data payload mismatches and improves coordination between frontend and backend development.

API base URLs and version prefixes are configurable through environment variables, allowing seamless switching between local and future hosted environments without requiring changes to application code.

6.3.3. Database Configuration

Modula uses PostgreSQL as its primary backend database, with all connection details managed through environment variables. Configuration parameters such as host, port, username, password, and database name are externalized to ensure that the application code remains independent of any specific database instance.

Instead of relying on an Object-Relational Mapping (ORM) framework, Modula uses raw SQL for database interaction. This decision provides greater transparency and control over database schema design, queries, and performance characteristics an important consideration for a transactional POS system. It also supports the academic objective of demonstrating explicit database design and data handling.

Database schema evolution is managed through a custom migration system. SQL migration files are stored in a dedicated migrations/ directory and executed via a single command during setup. Migrations are written to be idempotent, ensuring safe re-execution without data corruption. This simplifies onboarding for new developers and reduces setup complexity.

At the time of Capstone I, the database runs as a local instance. Containerized deployment using Docker is planned for Capstone II to improve portability, reproducibility, and deployment consistency across environments.

6.3.4. Dependency and Runtime Configuration

Dependency management and runtime configuration are standardized across the project to ensure reproducible builds and consistent development environments. The backend uses **pnpm** for dependency management, providing efficient installs and strict dependency resolution. The backend runtime is based on Node.js version 20.19, with TypeScript used to enforce static typing and improve maintainability.

The frontend uses the Flutter SDK with a fixed version constraint to ensure consistency across development machines. Flutter's dependency management system ensures that UI components, state management libraries (such as Riverpod), and networking tools remain aligned across platforms.

Development workflows are standardized through documented scripts and commands, reducing setup friction and minimizing environment-related issues. External infrastructure services, including Cloudflare R2 for media storage, are accessed through environment-based configuration and treated as replaceable components rather than tightly coupled dependencies.

Overall, Modula's configuration strategy supports the project's non-functional requirements for security, portability, and maintainability, while remaining flexible enough to accommodate future scaling and infrastructure changes beyond the scope of Capstone I.

6.4. Implementation of Features

This section describes the implementation of Modula's core and feature modules as realized in Capstone I. Each subsection explains the responsibility of the module, how it is implemented in the system, and how it interacts with other modules. While not all modules are production-complete, all described implementations are aligned with the defined Capstone I scope and architectural constraints.

6.4.1. Authentication and Authorization

Authentication and authorization form the entry point to the Modula POS system. The authentication component is responsible for validating user credentials and establishing an authenticated session, while the authorization mechanism determines which system capabilities are accessible to the user based on role and branch context.

In the current implementation, users authenticate using credential-based login. Role-based access control (RBAC) is enforced throughout the system, with roles such as

Administrator, Manager, and Cashier determining access to operational features such as sales, inventory, and reporting. Authorization checks are consistently applied at module boundaries, ensuring that protected actions cannot be executed without appropriate permission.

Although OTP-based verification and advanced identity workflows are planned for future phases, the existing authentication and authorization mechanism is sufficient to support secure role separation and branch-scoped access in Capstone I.

6.4.2. Tenant Context

In Modula, a tenant represents an independent business entity, such as a café, restaurant, or retail store. Each tenant operates as a fully isolated environment with its own users, policies, inventory, sales records, and reports. This design allows the system to support multiple businesses simultaneously without risk of data leakage or cross-tenant interference. Tenant isolation is enforced at the backend level and is consistently respected across all feature modules.

The Tenant Context module represents the ownership boundary of the system. A tenant corresponds to a single business entity that subscribes to Modula services and owns one or more branches. In Capstone I, tenant creation is system-driven and currently performed through developer tooling, with full automation deferred to a future billing engine.

The tenant context provides the root scope for policies, subscription limits, and audit visibility. All branches, users, and operational data are associated with a tenant, ensuring strict multi-tenant isolation. While tenants rarely change after creation, they serve as the foundational container for all other system modules.

6.4.3. Branch Management

In Modula, A branch represents a physical or logical outlet within a tenant, such as different store locations or service points. Branches are used to scope operational data including inventory levels, cash sessions, attendance records, and sales activity. While some configuration and reporting may occur at the tenant level, most day-to-day operations are branch-specific, reflecting real-world POS usage patterns.

Branch Management is a core module responsible for representing operational locations under a tenant. A branch is not created manually by users; instead, it is provisioned automatically by the system when a tenant is created or when an additional branch is added through subscription changes.

Each branch maintains its own operational identity, including name, address, and contact information. Branches can be placed into a frozen state by the system, typically due to subscription changes or administrative action. When a branch is frozen, operational actions such as sales, inventory updates, cash sessions, and attendance check-ins are blocked. However, historical data associated with the branch remains accessible for reporting and auditing purposes.

This separation between tenant and branch allows Modula to model real-world business operations accurately, where each location operates independently while remaining under a single organizational owner.

6.4.4. Policy and Configuration

The Policy and Configuration module defines operational rules that govern system behavior. Policies are resolved in the context of a branch, allowing different branches under the same tenant to operate with distinct configurations.

In Modula, policies are predefined by the system and provided with sensible default values. Users do not create new policies; instead, they modify existing configuration options in a manner similar to system settings on a mobile device. This design choice simplifies user experience, prevents misconfiguration, and ensures that all operational behavior remains within well-defined and supported boundaries.

Implemented policies include tax and currency handling, inventory behavior, cash session enforcement, and attendance constraints. For example, policies determine whether inventory is automatically deducted upon sale, whether a cash session is required before performing sales, and whether out-of-shift attendance requires managerial approval.

Policy evaluation occurs at runtime and is enforced consistently across modules. Branch freezing does not alter policy definitions but prevents policy-driven operations from being executed on frozen branches.

6.4.5. Sale and Order Management

Menu Management serves as the foundational configuration layer that defines what can be sold within the Modula POS system. Rather than operating as a standalone data management feature, the menu directly governs downstream operational behavior, including cart creation, pricing computation, discount application, inventory deduction, and reporting. As such, menu

configuration is treated as a controlled administrative activity whose effects propagate throughout the sales and inventory workflows.

In Modula, menu management is primarily performed by users with administrative privileges. Administrators define sellable menu items, organize them into categories, and configure optional modifiers that affect pricing and preparation. This configuration process establishes the canonical structure from which all sales operations derive. Cashiers and managers do not modify menu definitions during runtime; instead, they interact only with the menu as presented during the sales process, ensuring operational consistency and preventing unauthorized changes to pricing or product structure.

Menu items may exist independently of categories. When an item is created without an assigned category, it is automatically treated as uncategorized. This behavior ensures that incomplete configuration does not block sales operations while still signaling to administrators that further organization may be required. Categories function primarily as a navigational and organizational aid for both administrators and sales users, rather than as a strict functional dependency.

Modifiers and add-ons extend menu flexibility by allowing optional selections such as size, toppings, or preparation preferences that may affect the final price of an item. These modifiers are defined centrally and associated with specific menu items. During sales, selected modifiers contribute directly to the cart line price, forming part of the base calculation used for discounts, tax, and currency conversion. This design ensures that pricing remains transparent and predictable while supporting common food and beverage customization patterns.

Menu items may also be selectively assigned to branches. In multi-branch contexts, administrators can control item availability by enabling or disabling menu items per branch. This allows a single tenant to maintain a unified menu definition while tailoring availability based on branch-specific constraints such as equipment, inventory, or local offerings. Changes to branch assignments take effect immediately for new carts, but do not retroactively alter finalized sales.

Once menu items are actively used in finalized sales, Modula enforces immutability constraints to preserve historical accuracy. While administrators may update descriptive attributes such as names or images, changes that would alter the financial meaning of past transactions such as base prices or modifier pricing are constrained. This approach prevents

inconsistencies between historical sales records, receipts, and reports, aligning menu management with accounting and audit requirements.

From an operational perspective, the menu acts as the entry point to the sales workflow. Cashiers interact with the menu to create carts only when system policies permit sales activity, such as the presence of an active cash session. By separating menu configuration from runtime sales execution, Modula ensures that transactional behavior remains tightly controlled while still allowing administrators to evolve product offerings over time.

Overall, Menu Management in Modula exemplifies the system's design philosophy of configuration-driven operation. By treating the menu as a stable yet evolvable foundation, Modula enables flexible product definition without compromising transactional integrity, auditability, or operational clarity.

6.4.6. Menu Management

Menu Management serves as the foundational configuration layer that defines what can be sold within the Modula POS system. Rather than operating as a standalone data management feature, the menu directly governs downstream operational behavior, including cart creation, pricing computation, discount application, inventory deduction, and reporting. As such, menu configuration is treated as a controlled administrative activity whose effects propagate throughout the sales and inventory workflows.

In Modula, menu management is primarily performed by users with administrative privileges. Administrators define sellable menu items, organize them into categories, and configure optional modifiers that affect pricing and preparation. This configuration process establishes the canonical structure from which all sales operations derive. Cashiers and managers do not modify menu definitions during runtime; instead, they interact only with the menu as presented during the sales process, ensuring operational consistency and preventing unauthorized changes to pricing or product structure.

Menu items may exist independently of categories. When an item is created without an assigned category, it is automatically treated as uncategorized. This behavior ensures that incomplete configuration does not block sales operations while still signaling to administrators that further organization may be required. Categories function primarily as a navigational and organizational aid for both administrators and sales users, rather than as a strict functional dependency.

Modifiers and add-ons extend menu flexibility by allowing optional selections such as size, toppings, or preparation preferences that may affect the final price of an item. These modifiers are defined centrally and associated with specific menu items. During sales, selected modifiers contribute directly to the cart line price, forming part of the base calculation used for discounts, tax, and currency conversion. This design ensures that pricing remains transparent and predictable while supporting common food and beverage customization patterns.

Menu items may also be selectively assigned to branches. In multi-branch contexts, administrators can control item availability by enabling or disabling menu items per branch. This allows a single tenant to maintain a unified menu definition while tailoring availability based on branch-specific constraints such as equipment, inventory, or local offerings. Changes to branch assignments take effect immediately for new carts, but do not retroactively alter finalized sales.

Once menu items are actively used in finalized sales, Modula enforces immutability constraints to preserve historical accuracy. While administrators may update descriptive attributes such as names or images, changes that would alter the financial meaning of past transactions such as base prices or modifier pricing are constrained. This approach prevents inconsistencies between historical sales records, receipts, and reports, aligning menu management with accounting and audit requirements.

From an operational perspective, the menu acts as the entry point to the sales workflow. Cashiers interact with the menu to create carts only when system policies permit sales activity, such as the presence of an active cash session. By separating menu configuration from runtime sales execution, Modula ensures that transactional behavior remains tightly controlled while still allowing administrators to evolve product offerings over time.

Overall, Menu Management in Modula exemplifies the system's design philosophy of configuration-driven operation. By treating the menu as a stable yet evolvable foundation, Modula enables flexible product definition without compromising transactional integrity, auditability, or operational clarity.

6.4.7. Inventory Management

Inventory Management in Modula is designed to maintain accurate visibility of stock levels while supporting real-world operational practices in food and beverage businesses. Rather than functioning as an isolated feature, inventory management is tightly integrated with

menu configuration, sales execution, and policy enforcement. Its primary objective is to ensure that stock quantities reflect actual usage while remaining flexible enough to accommodate restocking, wastage, and manual adjustments.

Inventory configuration is performed by users with administrative privileges. Administrators define stock items that represent physical goods such as ingredients, packaging materials, or consumables. Each stock item is created with a base unit of measurement, which serves as the canonical unit for quantity tracking and deduction. This base unit enables consistent calculations when inventory is consumed through sales, even if restocking occurs in different packaging formats.

Stock items may be optionally organized into inventory categories. Similar to menu categorization, categories serve as an organizational and navigational aid rather than a strict requirement. When a stock item is created without a category, it is treated as uncategorized. This approach ensures that inventory setup can proceed incrementally without blocking operational readiness.

Inventory quantities are tracked at the branch level, reflecting the fact that physical stock is stored and consumed locally. Each branch maintains its own stock balance for each item, allowing multi-branch tenants to operate independently while still sharing a common menu and configuration model. Inventory views present branch-specific quantities, ensuring that staff have an accurate understanding of available stock at their location.

Modula supports inventory restocking through explicit restock operations. When stock is added, administrators record the quantity, unit, and optional metadata such as expiry date. If expiry tracking is enabled via policy, the system records stock in batches, allowing multiple lots of the same item each with different expiration dates to coexist. Internally, quantities are normalized to the base unit, ensuring consistency across restocking and consumption operations.

Inventory deduction occurs automatically during sale finalization when the relevant policy is enabled. Menu items may define ingredient mappings that specify how much of each stock item is consumed per sale. Upon finalizing a sale, the system deducts the appropriate quantities from branch inventory atomically with the sale transaction. This ensures that sales, inventory updates, and cash movements remain synchronized and auditable.

To support transparency and traceability, Modula maintains an inventory journal that records all stock-affecting events. Journal entries capture actions such as restocking, sale-based deductions, and manual adjustments. This historical record enables administrators to review stock movement over time, investigate discrepancies, and support reporting and audit requirements without modifying historical data.

Inventory data is also designed to work seamlessly with Modula's offline support. When operating offline, inventory-affecting actions are recorded locally and synchronized with the backend once connectivity is restored. Conflict resolution and reconciliation mechanisms ensure that the authoritative backend state remains consistent while still allowing uninterrupted operation at the branch level.

Overall, Inventory Management in Modula balances precision and practicality. By combining branch-level tracking, policy-controlled deduction, expiry awareness, and comprehensive journaling, the system provides reliable stock visibility while accommodating the operational realities of small and medium-sized businesses. This design ensures that inventory remains accurate, auditable, and aligned with the broader transactional workflows of the POS system.

6.4.8. Cash Session and Reconciliation

Cash Session and Reconciliation provide the operational controls needed to preserve cash integrity during day-to-day operation. In Modula, cash handling is implemented as an explicit session lifecycle that records an opening float, accumulates cash movements during the shift, and produces a closing reconciliation with variance. This approach reflects mature POS practice: cash accountability is enforced through structured shift procedures rather than informal "end-of-day" estimation.

A cash session is opened in the active **branch context** and is typically associated with the user operating sales on a given device. The session begins when the user enters the opening float (USD and/or KHR). This creates a traceable baseline and enables the system to compute the expected cash position throughout the shift. Importantly, Modula does not automatically terminate a session when the scheduled shift window ends; instead, the system allows a session to remain active to avoid disrupting in-progress operations and defers enforcement to approval workflows and reconciliation review.

The Sales module integrates directly with cash sessions through policy enforcement. When the branch policy **cashRequireSessionForSales** is enabled, Modula restricts the ability to draft a sale by preventing users from adding items to the cart until an active cash session exists. This eliminates uncontrolled growth of draft sales records while still allowing users to browse the menu. Once a sale is finalized, any cash tender amounts are recorded as cash movements attached to the active session, ensuring that cash totals are attributable to a specific session and actor.

During a session, Modula supports controlled "non-sale" cash movements. These include **paid out** (petty cash for supplies) and, where enabled, **manual adjustments** for exceptional correction. Such actions are gated by branch policies such as **cashAllowPaidOut**, **cashRequireRefundApproval**, and **cashAllowManualAdjustment**, and are recorded with a reason and actor identity to preserve traceability. Refund handling is similarly controlled: if refund approval is required, a cashier may initiate a refund request, but an Administrator or Manager must approve before reversal effects are applied.

Session close is a structured reconciliation step. The cashier inputs counted cash amounts, and Modula computes variance against the expected amounts derived from opening float and accumulated movements. Variance does not automatically imply wrongdoing; it may occur due to human counting error, small rounding effects, or operational realities. However, by recording variance explicitly and linking it to the session, Modula enables consistent oversight and produces reliable evidence for managerial review.

Manager and administrator intervention is intentionally constrained in Capstone I to preserve integrity. The system supports **force-closing** a session when needed (e.g., the cashier forgot to close) but does not support "taking over" an active session in a way that would blur responsibility across multiple actors. This preserves clear ownership of cash activity while still providing a practical operational escape hatch. The design remains future-ready: register-scoped sessions are identified as future work when Modula integrates with dedicated hardware environments.

6.4.9. Staff Management

Staff Management in Modula is responsible for controlling who can access the system, under which tenant and branch, and with what role and permissions. This module governs

organizational structure rather than operational activity, and it serves as a foundational layer that enables other modules such as Sales, Inventory, Cash Session, and Attendance to function correctly and securely.

Unlike Staff Attendance, which focuses on time-based presence and shift tracking, Staff Management is concerned with identity, membership, and authorization context. Its primary purpose is to ensure that only valid and properly assigned staff members can interact with the system, and that their access is constrained according to the tenant's subscription, branch configuration, and role definitions.

At the core of this module is the concept of tenant membership. Each user account may belong to one or more tenants, and within each tenant, the user is assigned a specific role (Administrator, Manager, or Cashier) and one or more branch assignments. This design supports real-world business scenarios in which a single individual may operate multiple businesses under the same account while maintaining strict separation of data and permissions between tenants.

The Staff Management module supports the controlled creation and onboarding of staff members through an invitation-based workflow. Administrators initiate staff creation by providing essential identity details such as first name, last name, role, and branch assignment. An invitation is then issued to the staff member, allowing them to complete account setup and gain access to the system. This approach prevents unauthorized access and ensures that every active staff member is explicitly approved by the tenant administrator.

Role assignment is a central responsibility of this module. Roles determine which features and actions a staff member may access within the system. Administrators have full control over tenant-wide configuration and oversight, Managers have elevated permissions within their assigned branches, and Cashiers are restricted to operational tasks such as sales execution and order handling. These role definitions are enforced consistently across all feature modules via the Authorization core module.

Staff Management also enforces staff quota constraints defined by the tenant's subscription. The system distinguishes between active and inactive staff members and applies soft and hard limits accordingly. Active staff count toward the tenant's usable capacity, while archived or deactivated staff are retained for audit and historical integrity but do not grant

operational access. This mechanism prevents abuse of staff limits while preserving traceability in audit logs and historical records.

Editing and deactivation of staff accounts are carefully controlled. Administrators may update staff assignments such as changing branch allocation or role but sensitive identity attributes and authentication credentials are managed through the Authentication and Authorization module rather than directly within Staff Management. Deactivation does not remove historical records; instead, it revokes access while maintaining referential integrity for past transactions, attendance logs, and audit entries.

From an architectural perspective, Staff Management integrates closely with several core modules. It depends on Authentication and Authorization for identity validation, Tenant and Branch Context for organizational structure, Policy and Configuration for enforcing limits and behaviors, and Audit Logging to record all staff-related changes. At the same time, it remains intentionally decoupled from operational modules, ensuring that business activity logic does not leak into organizational control.

In summary, the Staff Management module provides a clear and controlled framework for managing human access within Modula. By separating organizational membership and authorization from operational attendance and activity, Modula achieves stronger security, clearer modular boundaries, and a design that reflects the structure of real-world businesses. This separation is essential for scalability, auditability, and long-term maintainability of the POS system.

6.4.10. Staff Attendance

The Staff Attendance feature in Modula provides a structured and policy-driven mechanism for recording staff presence at individual branches. Its primary objective is to ensure accountability and transparency in daily operations while remaining simple enough for small and medium-sized food and beverage businesses. The design intentionally avoids excessive complexity and focuses on reliable record-keeping rather than advanced workforce analytics.

In Modula, attendance is recorded through explicit check-in and check-out actions performed by staff members. Attendance is always evaluated within a branch context, meaning that each staff member can only check in at the branch to which they are currently assigned.

This design prevents ambiguous or overlapping attendance records across branches and reinforces operational clarity in multi-branch environments.

The behavior of staff attendance is governed by policies defined in the Policy and Configuration module. These policies determine whether early check-ins are permitted, whether out-of-shift check-ins require approval, and whether attendance should be linked to other operational activities such as cash sessions. When a staff member attempts to check in, the system evaluates the applicable policies before allowing the action to proceed. If the check-in complies with all policy conditions, an immutable attendance record is created immediately. If the check-in violates a policy constraint such as occurring outside an assigned shift an approval request is generated instead.

Approval workflows play a key role in balancing operational flexibility with managerial control. When out-of-shift approval is required, a manager or Administrator may review pending requests and either approve or reject them. Approval activates the attendance record without altering the original timestamps, while rejection prevents the creation of an attendance entry altogether. This ensures that attendance data remains trustworthy and auditable, even when exceptions occur.

Role-based access is strictly enforced within the attendance feature. Cashiers and Managers may view their own attendance history, allowing them to track working hours and past check-ins. Managers additionally have visibility into attendance records for staff within their assigned branch. Administrators have global visibility across all branches, enabling oversight at the tenant level. Importantly, neither Managers nor Administrators are permitted to edit or delete attendance records; attendance data is treated as immutable once recorded, reinforcing its reliability as an operational and audit artifact.

The Staff Attendance feature is designed with offline-first operation in mind. Check-in and check-out actions performed while offline are stored locally on the client device and synchronized with the backend once connectivity is restored. During synchronization, policy enforcement and conflict resolution rules are applied to ensure consistency with server-side records. This approach allows Modula to function reliably in environments with unstable or intermittent internet connectivity, which is common in real-world POS deployments.

By limiting the scope of attendance functionality in Capstone I to essential features such as recording presence, enforcing policies, and supporting approvals Modula avoids unnecessary complexity while laying a strong foundation for future enhancements. Advanced

capabilities such as GPS verification, automated reminders, or attendance analytics are intentionally deferred to later phases. This staged approach aligns with the project's modular philosophy and ensures that the attendance feature remains consistent, auditable, and easy to use within the overall POS system.

6.4.11. Discount Management

Discount Management in Modula is responsible for defining and maintaining commercial pricing adjustments that apply to sales transactions within a specific branch context. This feature module is intentionally separated from both the Policy module and the Sale module to preserve clear system boundaries: policies govern operational behavior, while discounts define business-facing pricing rules. The Discount Management module provides a controlled and predictable mechanism for applying price reductions without introducing ambiguity into the sales workflow.

In Capstone I, all discounts are branch scoped. A discount defined for one branch applies only to sales conducted within that branch and has no effect on other branches under the same tenant. This design aligns with real-world operational practices in food and beverage businesses, where promotions and pricing strategies often vary by location. Cross-branch discount sharing and centralized discount campaigns are deliberately excluded from the current scope and reserved for future development phases.

The discount model in Modula is intentionally minimal and focused. Only percentage-based discounts are supported in Capstone I, reducing complexity and ensuring pricing transparency for both staff and customers. Discounts may be defined at two levels: item-level discounts that apply to specific menu items, and branch-wide discounts that apply to all eligible sales within a branch. Fixed-amount discounts, coupon codes, customer-specific pricing, and time-based promotions are not included in this phase to avoid unnecessary edge cases and operational confusion.

Discount configuration and maintenance are restricted to administrative users. Administrators have full control to create, update, enable, or disable discounts within a branch. Managers and cashiers do not have permission to modify discount settings and are limited to viewing discount effects indirectly through the sales interface. This role separation ensures that pricing authority remains centralized while preserving operational simplicity for frontline staff.

The interaction between Discount Management and the Sale module is strictly one-directional. During the sale process, the Sale module retrieves all applicable discounts for the active branch and computes discounted prices at the cart stage. Once a sale is finalized, the resolved discount values are locked in and persisted as part of the sale record. Subsequent changes to discount configurations do not retroactively affect completed sales or existing orders. This lock-in mechanism preserves financial accuracy, supports auditability, and prevents disputes arising from post-transaction price changes.

By separating discount definition from transaction execution, Modula achieves a clear division of responsibility between feature modules. Discount Management focuses on expressing commercial intent, while the Sale module focuses on executing transactions reliably and consistently. This design reflects Modula's broader architectural philosophy of modularity and explicit responsibility boundaries, while remaining practical and appropriate for the scope of Capstone I.

Overall, the Discount Management feature balances simplicity and control. Its branch-scoped, percentage-only design minimizes operational risk, supports transparent pricing, and aligns with the needs of small and medium-sized F&B businesses. At the same time, the module establishes a solid foundation for more advanced discounting capabilities in future phases without requiring fundamental redesign.

6.4.12. Receipt and E-Receipt

The Receipt and E-Receipt module in Modula are responsible for presenting a finalized and immutable record of completed sales. Rather than acting as a configuration-heavy or transaction-processing component, this module serves as a read-only representation of financial outcomes that have already been validated and committed by the Sale and Order Management module. Its primary objective is to provide clear, accurate, and consistent transaction records for both customers and business operators, while maintaining strict financial integrity.

In Modula, receipts are generated only after a sale has been finalized. At this point, all relevant computations such as item pricing, modifier costs, discounts, tax application, currency conversion, and rounding have already been resolved and locked. The receipt therefore represents a snapshot of the final transaction state. Once generated, a receipt cannot be edited or altered. Any correction to a completed transaction must follow the formal voiding workflow

defined in the Sale and Order Management module, ensuring that financial adjustments are auditable and do not compromise data integrity.

The module supports electronic receipts (eReceipts) as the primary output in Capstone I. Each receipt displays essential transaction details, including purchased items, applied discounts, tax amounts, payment method, currency breakdown, and final totals. Receipts are tightly coupled with orders; users access an eReceipt by selecting a finalized order from the order list. This order-centric access model reflects common practices in food and beverage POS systems and simplifies navigation by avoiding a separate receipt management interface.

Receipt presentation also incorporates branch-specific customization in a controlled and limited form. While core business information such as brand name, branch name, address, and contact details is managed by the Tenant and Branch Context module, the Receipt and eReceipt module allows each branch to define a customizable footer. This footer may include content such as a thank-you message, return or refund notes, or promotional text. Footer customization is scoped at the branch level and applied consistently to all receipts generated for that branch, ensuring uniform branding without introducing per-transaction variability.

Importantly, the Receipt and eReceipt module does not participate in business rule enforcement or data mutation. It consumes finalized data from upstream modules and renders it in a customer-facing format. This separation of concerns aligns with Modula's modular design philosophy and reduces coupling between financial logic and presentation logic. Access to receipts is logged through the Audit Logging module, providing traceability without exposing sensitive data to modification.

In Capstone I, physical receipt printing and hardware integration are explicitly out of scope. The focus on eReceipts allows the system to deliver essential functionality while remaining platform-agnostic and suitable for web-based operation. Support for thermal printers and dedicated POS hardware is planned for future phases, where the receipt module can be extended without altering its core responsibilities.

Overall, the Receipt and eReceipt module reinforces Modula's emphasis on transactional clarity, immutability, and modularity. By treating receipts as derived, read-only artifacts of finalized sales, the system ensures consistency across customer-facing views, administrative workflows, and reporting outputs, while maintaining a clean separation between financial processing and presentation.

6.4.13. Reporting

The reporting module in Modula provides administrators with structured visibility into business operations while preserving the integrity and auditability of transactional data. Rather than introducing a separate reporting datastore, the system derives all reports directly from authoritative records produced by the sales, cash session, inventory, and policy modules. This design ensures that reported values always reflect the actual state of the system and avoids inconsistencies caused by duplicated or denormalized reporting data.

In Capstone I, reporting is deliberately scoped to a **single-branch** context and is accessible only to users with the Administrator role. This decision aligns with the project's incremental delivery strategy and reflects the operational reality of small food and beverage businesses, which typically review performance at the branch level. Cross-branch aggregation and advanced analytical views are intentionally deferred to later phases in order to reduce complexity and maintain clarity during initial implementation.

Sales reporting in Modula summarizes finalized transaction data, including item-level sales, total revenue, applied taxes, and discounts. A key design consideration is the treatment of voided and void-pending transactions. Sales marked as VOID_PENDING remain visible in reports but are clearly labeled as provisional and excluded from confirmed revenue calculations. This ensures transparency while preventing premature removal of financial records. Once a void request is approved, the sale transitions to VOIDED and is fully excluded from revenue and cash totals. This approach mirrors established practices in mature POS systems, where reversals are approval-based and must remain traceable for audit purposes.

Cash-related reporting is tightly integrated with the cash session module through industry-standard X and Z reports. X reports provide an in-session snapshot of cash activity without closing the session, while Z reports are generated only upon session closure and represent finalized records for reconciliation. Z reports are immutable once created, reinforcing the principle that completed cash sessions cannot be altered retroactively. Both report types respect the tenant's currency configuration, VAT settings, and rounding policies.

All reporting views in Modula are strictly read-only. The reporting interface does not expose any functionality to modify sales, cash movements, or historical records. This immutability is a deliberate design choice that protects financial correctness and prevents reporting screens from becoming indirect mutation paths. Every report access is logged through

the audit logging module, ensuring that administrative visibility is itself traceable and accountable.

By limiting reporting in Capstone I to essential summaries with strong accounting safeguards, Modula establishes a reliable foundation for operational oversight. This design balances immediate business needs with long-term extensibility, allowing future phases to introduce cross-branch analytics, scheduled reports, and advanced visualization without compromising the integrity of core financial data.

6.4.14. Audit Logging

Audit Logging in Modula serves as a foundational support feature that ensures accountability, traceability, and operational transparency across the entire POS system. Rather than functioning as a developer-oriented debugging mechanism, the audit logging feature is explicitly designed as an administrative capability accessible to tenant administrators. Its primary purpose is to provide a reliable and immutable record of critical system actions performed by users across different modules.

In Modula, audit logs are system-generated and read-only. Every significant action such as user authentication events, sale finalization, sale void requests and approvals, inventory adjustments, cash session openings and closures, policy changes, and attendance check-ins is automatically recorded by the system. These logs cannot be manually created, edited, or deleted by any user role, ensuring the integrity and trustworthiness of the recorded data.

Access to audit logs is restricted by role. Tenant administrators are granted full read access to audit logs across all branches under their tenant, allowing them to review historical actions for oversight and governance purposes. Managers may be granted limited or view-only access depending on policy configuration, while cashiers have no access to audit logs. This role-based access control ensures that sensitive operational data is visible only to authorized users.

From a functional perspective, the audit logging feature allows administrators to:

- Review actions performed by staff members across modules
- Trace the origin and approval flow of sensitive operations such as sale voids or cash adjustments
- Investigate discrepancies in inventory, cash reconciliation, or attendance
- Support dispute resolution by providing a verifiable action history

Audit logs are presented through a dedicated administrative interface that supports filtering and searching by criteria such as date range, module, action type, user, and branch. This structured presentation enables efficient analysis without overwhelming the administrator with raw system data.

Importantly, audit logging is deeply integrated with Modula's modular architecture. Each feature module emits audit events in a consistent format when key actions occur, allowing the audit system to capture cross-module activity without tightly coupling modules to one another. This design reinforces modularity while ensuring comprehensive coverage of system behavior.

Overall, the Audit Logging feature strengthens Modula's reliability and trust model by ensuring that all critical operations are observable, attributable, and historically preserved. By exposing audit logs as an administrative capability rather than a hidden technical detail, Modula aligns with best practices observed in mature commercial POS systems and meets real-world operational needs for transparency and control.

6.4.15. Sync and Offline Support

Reliable operation under unstable network conditions is a critical requirement for Point-of-Sale systems, particularly in small and medium-sized business environments where connectivity cannot be guaranteed at all times. To address this operational constraint, Modula implements Sync and Offline Support as a core system capability that enables continuous operation while preserving data integrity and consistency.

In Modula, Sync and Offline Support is designed as an infrastructure-level function rather than a user-facing feature. It operates transparently in the background and primarily serves cashier workflows, ensuring that essential operations such as sale creation, order updates, cash session actions, and attendance check-ins can continue even when the application is temporarily offline.

From an architectural perspective, offline support is implemented using IndexedDB on the client side, while PostgreSQL remains the authoritative system of record on the backend. When network connectivity is unavailable, user actions are recorded locally as structured operations in IndexedDB. Each operation is assigned a unique client-generated identifier to support idempotent processing. Once connectivity is restored, these queued operations are synchronized with the backend in a controlled and sequential manner.

The synchronization process follows a strict responsibility separation. The client is responsible for detecting connectivity changes, persisting offline actions locally, and replaying them when possible. The backend validates each incoming operation and ensures that duplicate submissions do not result in inconsistent state changes. This approach prevents issues such as duplicate sales, repeated inventory deductions, or inconsistent cash movements, even if synchronization is interrupted and retried multiple times.

Sync and Offline Support integrate closely with several core workflows. In the Sales module, finalized sales created offline are synchronized and reconciled with inventory deduction, cash session records, and reporting data once connectivity is restored. In the Cash Session module, cash movements recorded during offline operation are queued and later applied atomically. Similarly, attendance records created while offline are synchronized without allowing retroactive modification, preserving auditability.

From a user experience perspective, Modula prioritizes non-disruptive behavior. Cashiers are not blocked from performing allowed actions when the system enters offline mode. Instead, the application provides clear visual indicators to communicate offline status and synchronization progress, while maintaining consistent user interaction patterns. This design minimizes operational friction during peak business hours and reduces the cognitive load on users.

The scope of Sync and Offline Support in Capstone I is intentionally limited. Advanced features such as manual conflict resolution, multi-device offline reconciliation, administrative control over synchronization queues, and offline inventory restocking are explicitly deferred to future development phases. By constraining the scope, the implementation remains focused, reliable, and aligned with the project's modular architecture.

Overall, Sync and Offline Support reinforce Modula's design philosophy of operational resilience and modular responsibility separation. By enabling offline-first behavior without compromising data integrity, this capability ensures that Modula remains practical and dependable in real-world POS environments, even under imperfect network conditions.

VII. CONCLUSION

7.1. Completed Features

This section summarizes the features and modules implemented during Capstone I, categorized according to their level of completion within the defined project scope. It is important to note that "implemented" in this context refers to functionality completed and demonstrable within the Capstone I scope, rather than production-level readiness. Several modules were intentionally implemented with functional limitations or deferred for future phases due to time, resource, and scope constraints.

7.1.1. Core Modules

- **Authentication and Authorization:** Authentication and authorization were implemented using a role-based access control model that supports Administrators, Managers, and Cashiers. Phone-based login and session handling were completed, and users are correctly scoped to tenants and branches. Multi-tenant support per account was implemented at the system level. However, advanced security mechanisms such as OTP-based verification were designed but not implemented within the Capstone I timeframe.
- **Tenant and Branch Context:** The tenant and branch context was implemented to support multi-branch operations under a single tenant, with branch-scoped data access and permissions. The system supports multiple tenants under a single account model. Automatic tenant creation triggered by a billing or subscription engine was not implemented; instead, tenant provisioning is currently developer-driven, which aligns with the academic scope of Capstone I.
- **Policy and Configuration:** The policy and configuration module was fully implemented within Capstone I. It provides centralized control over operational behaviors such as VAT application, currency conversion and rounding, inventory deduction behavior, cash session enforcement, and attendance rules. Policies are applied consistently across relevant modules and are configurable at runtime, forming a core mechanism for adapting system behavior to different operational needs.
- **Audit Logging:** Audit logging was designed as a cross-cutting core module intended to capture critical system actions across all feature modules. Partial implementation was completed during Capstone I, and the remaining coverage was finalized during the

reporting phase. The module provides the foundation for traceability, accountability, and future compliance-related extensions.

- **Sync and Offline Support:** Full offline synchronization was deferred to future work. While client-side storage mechanisms were prepared and IndexedDB was integrated at the frontend level, end-to-end offline operation and conflict resolution were not completed within Capstone I due to complexity and time constraints.

7.1.2. Feature Modules

- **Sales and Order Management**: Sales and order management were fully implemented and represent the operational core of the system. The module supports cart-based sales, multi-currency cash handling, policy-enforced checkout, order lifecycle management, and approval-based sale voiding. Sales transition into orders with controlled state changes, ensuring financial integrity and operational traceability.
- **Menu Management**: Menu management was implemented with support for menu items, categories, modifiers, and branch-level assignment. The module enables structured menu configuration suitable for food and beverage operations. Although recipe-to-inventory mapping exists, automatic inventory deduction based on recipes was not enabled during Capstone I.
- **Inventory Management**: Inventory management was implemented with support for stock items, branch-level stock tracking, restocking, expiry tracking, and inventory journaling. The module supports accurate manual inventory updates and reporting. Automatic deduction based on menu recipes was intentionally deferred, pending further integration with the menu module.
- **Staff Management**: Staff management was implemented to support staff creation, role assignment, branch association, and staff listing. Credential management actions such as password reset were intentionally excluded to maintain clear separation between staff management and account-level authentication concerns.
- **Staff Attendance**: The staff attendance module was fully implemented, providing check-in and check-out functionality, shift-based enforcement, and approval workflows for out-of-shift attendance. Attendance data is branch-scoped and integrates with policy configuration to enforce operational rules.
- **Reporting**: Basic reporting functionality was implemented to support sales reports, inventory summaries, and cash session reports. Reports correctly handle pending void

transactions and finalized data states. Advanced analytics and cross-branch aggregation were deferred to future phases.

- **Discount Management**: Discount management was specified and designed but not implemented during Capstone I. The module was deferred due to prioritization of core transaction and operational stability.
- **Receipt and Electronic Receipt**: Receipt and electronic receipt functionality was partially implemented at the design and UI level. Static user interface designs and interaction flows were created using Figma; however, full backend integration and dynamic receipt rendering were deferred.
- **User Interface and User Experience**: UI and UX implementation progressed iteratively throughout Capstone I. Detailed wireframes were created, but final interaction behaviors were refined during frontend development through manual testing. Feedback from implementation informed adjustments to both UX flow and module specifications.
- **Infrastructure and Deployment**: Production infrastructure, cloud deployment, and containerization were not implemented during Capstone I. The system currently runs in a local development environment, with infrastructure planning deferred to subsequent phases.

In summary, for Capstone I we successfully delivered the foundational core and feature modules required for a functional, modular POS system. Several components were intentionally implemented with limitations or deferred to future phases to prioritize architectural clarity, transactional correctness, and academic feasibility within the project's constraints.

7.2. Difficulties

The development of Modula POS presented multiple challenges that extended beyond feature implementation and exposed the practical realities of building a production-oriented system within an academic timeframe.

One of the primary difficulties was managing the complexity of a modular system. Although Modula was designed with modular architecture principles, the modules were not entirely independent. Core modules such as Authentication, Policy, Sales, Inventory, Cash Session, and Attendance shared common domain concepts and operational dependencies. As a result, changes made to one module such as adjusting policy scope from tenant-wide to branch-

wide often required corresponding updates across several other modules. This interdependency increased development overhead and made system-wide consistency harder to maintain.

Another major challenge involved maintaining alignment between documentation and implementation. Modula relied heavily on module specifications (mod specs) to guide development. However, during implementation and manual testing, certain assumptions defined in early specifications proved impractical or misaligned with real operational workflows. Updating mod specs while simultaneously refactoring backend logic and frontend behavior required continuous effort. Ensuring consistency between documentation, API contracts, database schema, and frontend data models became an ongoing challenge throughout the project.

The project also faced integration difficulties between frontend and backend development. Although frontend and backend teams worked in parallel using agreed-upon specifications, integration often revealed mismatches in payload structures, naming conventions, and lifecycle expectations. Complex workflows such as sale finalization, inventory deduction, cash session enforcement, discount resolution, and offline synchronization required multiple iterations to stabilize across system layers.

Additionally, the team encountered challenges related to limited experience and time constraints. Several aspects of the project, including modular backend architecture, offline-first design, transactional integrity, and policy-driven behavior, required learning beyond prior coursework. The learning curve consumed a significant portion of the available timeline, making it difficult to progress according to the original schedule.

Overall, these difficulties highlighted the gap between theoretical system design and practical implementation. While challenging, they provided valuable exposure to real-world software engineering constraints that are rarely encountered in smaller or purely academic projects.

7.3. Lessons Learned

The Modula POS project offered important lessons in software engineering, system architecture, and project management that extend beyond technical execution.

One key lesson was the importance of realistic scope definition. The initial scope of the project was ambitious, aiming to deliver a wide range of features and architectural capabilities within a limited timeframe. In practice, the available time and human resources were

insufficient to fully implement all planned components at the desired level of maturity. This experience emphasized the necessity of prioritization, incremental delivery, and clear separation between minimum viable functionality and future enhancements.

Another significant lesson concerned modularity and dependency management. While modular architecture improves clarity and maintainability, it does not eliminate inter-module dependencies. The project demonstrated that core concepts such as policies, branch context, and transactional data naturally span multiple modules. Effective modular design therefore requires continuous coordination and architectural discipline, rather than assuming complete isolation between components.

The team also learned the importance of iterative and adaptable documentation. Early module specifications were written with excessive detail, including assumptions about specific API endpoints and frontend behavior. This rigidity made the documentation difficult to maintain as implementation realities evolved. Transitioning to higher-level specifications focusing on purpose, use cases, requirements, and acceptance criteria proved more effective and reduced friction between design and development.

From a human and organizational perspective, the project highlighted the challenge of accurately estimating team capability and learning curves. While team members possessed foundational technical knowledge, implementing a real-world POS system introduced unforeseen challenges. Considerable time was spent learning new tools, debugging integration issues, and refining architectural decisions. This reinforced the importance of factoring skill development and experimentation into project planning.

Despite these challenges, the project delivered substantial learning outcomes. Team members gained deeper experience in modular system design, cross-layer integration, API versioning, offline-first considerations, and collaborative development practices. The difficulties encountered during the project ultimately contributed to stronger technical judgment and a more realistic understanding of software engineering in practice. Modula POS thus served not only as a functional capstone project but also as a valuable simulation of real-world system development, where adaptability, communication, and iterative improvement are essential to success.

7.4. Perspectives

This section outlines the potential future directions of the Modula POS system beyond the scope of Capstone I. These perspectives do not represent immediate implementation

commitments, but rather identify feasible extensions enabled by the current architectural foundation, modular design, and technology choices. The discussion reflects both technical opportunities and lessons learned during development, and it positions Modula for continued evolution as a scalable POS platform.

7.4.1. Functional Extensions

- One of the primary areas for future development is the expansion of functional features that were intentionally deferred or only partially implemented in Capstone I. In particular, the sync and offline support module represents a significant opportunity for enhancement. While the system architecture already anticipates offline operation through client-side storage and synchronization mechanisms, future work may include more advanced conflict resolution strategies, background synchronization, retry queues, and improved user feedback during network disruptions. These enhancements would strengthen Modula's suitability for environments with unstable connectivity.
- Another key extension involves the discount management module, which is designed but not fully implemented in Capstone I. Future iterations may introduce more comprehensive branch-scoped discount configurations, improved rule visualization for administrators, and tighter integration with reporting. Similarly, receipt and eReceipt customization can be extended beyond static layouts to support dynamic templates, richer branding options, and localized formatting per branch.
- Reporting functionality may also evolve toward more advanced operational analytics, such as historical trend analysis, performance indicators, and comparative summaries across time periods. These enhancements would provide business owners with deeper insights into sales performance and inventory movement while remaining focused on operational decision-making rather than complex business intelligence.

7.4.2. Architectural and Infrastructure Evolution

- From an architectural perspective, Modula is designed to remain hosting-provider agnostic, allowing future deployment strategies to be evaluated independently of application logic. In later phases, the system may be containerized using technologies such as Docker to support consistent deployment across environments. This would simplify production setup, scaling, and maintenance while preserving the modular monolithic architecture adopted in Capstone I.

- As usage grows, further refinements to database migration strategies, API versioning, and event-driven integration may be introduced. The existing separation between feature modules and shared platform services provides a strong foundation for scaling individual components without compromising overall system coherence. These architectural evolutions aim to support reliability and maintainability as Modula transitions from an academic prototype to a long-term software product.

7.4.3. Security and Reliability Improvements

- Although Modula implements core authentication and authorization mechanisms in Capstone I, several security-related enhancements are planned for future phases. These include OTP-based verification for sensitive account actions, improved rate limiting to prevent abuse, and additional safeguards against denial-of-service scenarios. Such measures would enhance system resilience while aligning with common security practices in production POS systems.
- Reliability improvements may also involve more comprehensive audit coverage, automated backup strategies, and enhanced monitoring of critical operations such as cash reconciliation and inventory updates. These enhancements are supported by the existing audit logging and policy-driven design, which were deliberately structured to accommodate future security extensions without major refactoring.

7.4.4. Product and Platform Growth

- Finally, Modula's long-term vision includes growth beyond its initial web-based deployment. The use of Flutter positions the system for mobile-native expansion, enabling dedicated tablet or smartphone POS applications without rewriting core frontend logic. This supports multi-device usage scenarios and aligns with modern POS deployment trends.
- Future development may also explore controlled hardware integration, such as receipt printers or cash drawers, building upon the existing cash session and order workflows. Additionally, improvements to multi-tenant account support may allow a single user account to manage multiple businesses more seamlessly, addressing limitations identified during Capstone I.
- Overall, these perspectives demonstrate that Modula's current implementation is not an isolated academic exercise but a deliberately structured foundation for continued development. By prioritizing modularity, clarity of responsibilities, and policy-driven

behavior, the system is well-positioned to evolve incrementally while preserving consistency and maintainability.

REFERENCES

[1] K. O. S. Sai, "An analysis of point of sale systems physical configurations and security measures in Zimbabwean SMEs," IRA International Journal of Education and Multidisciplinary Studies, vol. 6, no. 2, p. 181, Mar. 2017

[2] R. Kotha, "The Evolution of Point-of-Sale Systems from Traditional to Mobile Solutions," Journal of Artificial Intelligence Machine Learning and Data Science, vol. 1, no. 4, pp. 1181–1185, Dec. 2023

[3] A. Ojeda, "Point-of-sales systems in food and beverage industry: Efficient technology and its user acceptance," 2017. [Online]. Available: https://www.researchgate.net/profile/Angel-Ojeda-4/publication/322926135_Point-Of-Sales_Systems_in_Food_and_Beverage_Industry_Efficient_Technology_and_Its_User_Acceptance/links/5a7711b1a6fdccbb3c097e02/Point-Of-Sales-Systems-in-Food-and-Beverage-Industry-Efficient-Technology-and-Its-User-Acceptance.pdf

[4] Square, "Point of Sale software pricing," Square, 2025. [Online]. Available: https://squareup.com/us/en/point-of-sale/software/pricing

[5] Toast, "Point of Sale," Toast, 2025. [Online]. Available: https://pos.toasttab.com/products/point-of-sale

[6] Lightspeed, "Retail POS," Lightspeed, 2025. [Online]. Available: https://www.lightspeedhq.com/pos/retail/

[7] TechnologyAdvice, "POS Features: Essential Functions for Modern Point-of-Sale Systems," TechnologyAdvice, 2025. [Online]. Available: https://technologyadvice.com/blog/sales/pos-features/

[8] Olitt, "Square Online: Advantages and Disadvantages – Is It Right for Your Business?," Olitt, 2025. [Online]. Available: https://olitt.com/blog/square-online-advantages-and-disadvantages-right-for-your-business/

[9] Deliverect, "Toast POS: Everything You Need to Know," Deliverect, 2025. [Online]. Available: https://www.deliverect.com/en/blog/pos-systems/toast-pos-everything-you-need-to-know

[10] Crazy Egg, "Toast POS Review: Features, Pros & Cons," Crazy Egg, 2025. [Online]. Available: https://www.crazyegg.com/blog/toast-pos-review/

[11] Champei POS, "Features," Champei POS, 2025. [Online]. Available: https://champeipos.com/features/

[12] M-POS, "Software," M-POS, 2025. [Online]. Available: https://www.m-pos.cc/en/software

[13] Triare, "Modular Architecture in Software: Benefits and Insights," Triare, 2025. [Online]. Available: https://triare.net/insights/modular-architecture-software/

[14] Wikipedia, “Modular design,” Wikipedia, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Modular_design
[15] NordVPN, “Modular Software Architecture,” NordVPN, 2025. [Online]. Available: https://nordvpn.com/cybersecurity/glossary/modular-software-architecture/