Vendomatic Project

Architecture Notebook

Version 1.2

Prepared By: Code Busters

Revision History

Version	Description	Author	Date
0.1	● Draft	Kadir Kılıçoğlu	2023-04-12
0.2	● Architecturally Significant Requirements are added. ● Architectural Mechanisms are added. ● Domain model is added under Section-7 Key Abstractions	Tuğçe Sözer	2023-04-23
0.3	● Review and format	Ezgi Özkan	2023-04-24
1.0	● Add logical and data views	Kadir Kılıçoğlu	2023-05-11
1.1	● Add more constraints and decisions ● Add more diagrams	Kadir Kılıçoğlu Tuğçe Sözer	2023-06-01
1.2	● Review and format	Ezgi Özkan	2023-06-04

Table of Contents

1. Purpose

This document describes the architectural philosophy, decisions, constraints, justifications, significant elements, and overarching aspects of the Vendomatic system, a vending machine management and customer interaction platform that includes a web application for vending machine owners/operators and a mobile application for customers.

The architecture notebook will serve as a reference for the development team throughout the development process, helping to ensure that the application is designed and implemented according to best practices and industry standards. Additionally, it will provide a foundation for future development and maintenance, enabling the platform to adapt and evolve as business needs and technology requirements change over time.

2. Architectural Goals and Philosophy

The architectural goals and philosophy of the Vendomatic system are centered around addressing specific issues and considerations that drive the design decisions.

Scalability: The architecture must be designed to handle increasing user loads efficiently, ensuring the system remains performant even during peak usage periods. This includes considering the deployment of services across multiple servers or using cloud-based infrastructure to accommodate growth.

To address scalability concerns, Vendomatic backend consisting of the database and the API for both mobile and web applications should be hosted on the cloud. Though, the services should be hosted on a single server until reaching a broader audience to ease faster deployment during Elaboration and Construction phases.

Integration: The architecture should be built with a focus on integrating seamlessly with external systems and services, such as payment gateways, social media login providers, and third-party APIs. This will enable the system to leverage existing technologies and simplify the implementation of new features.

Security: The architecture must prioritize the protection of sensitive user data and transactions. This includes employing secure communication channels, proper data encryption, and adhering to best practices for handling user authentication and authorization.

Maintainability: The architecture should be designed for long-term maintenance, ensuring that the system can be easily updated and extended as business requirements evolve. This includes the use of well-defined interfaces, clean code practices, and comprehensive documentation to support future development efforts.

Platform Independence: The architecture should minimize hardware dependencies and enable the system to run on different platforms and environments, ensuring that it remains flexible and adaptable to various deployment scenarios.

By addressing these critical concerns, the architecture aims to provide a solid foundation for the Vendomatic system, ensuring that it meets the demands of its users while remaining adaptable to future changes and enhancements.

3. Assumptions and Dependencies

3.1. The Vendomatic Web Application and Mobile Application will communicate with a centralized backend.
3.2. The vending machines will be equipped with the Vendomatic adapter for remote management.
3.3. The system depends on the user's device providing accurate location data to enable features such as searching for nearby vending machines or filtering based on proximity. This dependency requires the user to grant the necessary location permissions and assumes that their device's GPS functionality is functioning correctly.
3.4. The development team is familiar with modern web and mobile development technologies.
3.5. The system will rely on third-party services for payment processing, map integration, and social authentication.
3.6. The system assumes that users have access to a reliable internet connection to interact with the Vendomatic Mobile Application and access the various features and functionalities. This dependency may affect the system's performance in areas with limited or unstable network connectivity, necessitating the implementation of offline capabilities or caching mechanisms where possible.
3.7. The system assumes that it complies with any applicable regulations and industry standards, such as data protection and privacy laws.

4. Architecturally Significant Requirements

4.1. Secure user authentication and role-based access control for different user types shall be provided (vending machine owners, operators, customers).
4.2. Owners shall be registered during the setup phase, and a single Vendomatic Software on the web platform shall have a single owner.
4.3. Social logins shall be supported only by the Vendomatic Mobile Application.
4.4. Username shall be the email of the user.
4.5. Passwords shall be at least 8 characters containing at least one letter and one number.
4.6. All inputs shall be both client-side and server-side validated.
4.7. Access to the system shall use OAuth 2.0, OIDC supported JWT token and Firebase as IdAAS to authenticate all users.
4.8. No password will be kept by the system, hence passwords won’t be tracked.

5. Decisions, Constraints and Justifications

1	**Decision**	Use a modular monolith architecture to enable scalability, maintainability, and flexibility.
	Justification	The Vendomatic system will have multiple components with different responsibilities, and a modular monolith architecture will allow them to evolve independently while supporting a less complex deployment strategy. One alternative is the pure microservice architecture, but considering its deployment complexity and lack of distributed teams, we eliminated this option.

		Another alternative, the vanilla monolith, is hard to maintain and doesn't fit well into iterative life cycles, thus we eliminated this option as well..
2	**Constraint**	All data transmitted between the web/mobile applications and backend services must be encrypted.
	Justification	Ensuring the privacy and security of user data is a priority. Though encryption slightly degrades the performance, the tradeoff of using unencrypted traffic may have a huge impact by resulting in a vulnerable model. Hence, encryption is preferred over performance.
3	**Decision**	Use a responsive design for the web application.
	Justification	This ensures that the web application will be usable on various devices and screen sizes. Responsive design has become the mainstream recently. Working with a strict layout may be easier, but since there are various resolutions and screen types, we skipped this option.
4	**Constraint**	The mobile application must be developed for Android and iOS platforms.
	Justification	To reach a wider customer base. Otherwise, with a single platform target, Vendomatic wouldn't reach millions of users of the ignored platform.
5	**Decision**	The data store for the common API would be PostgreSQL.
	Justification	We have decided to use PostgreSQL as the data store for the common API due to several reasons. PostgreSQL is fully ACID-compliant (Atomicity, Consistency, Isolation, Durability), ensuring data integrity and consistency even during system failures. It has many features aimed at developers, such as advanced indexing, arrays, store, and JSON/JSONB, which can simplify data handling and operations. PostgreSQL has strong performance capabilities and is renowned for its efficiency and speed, crucial attributes for an API that will handle a high volume of requests. Furthermore, PostgreSQL is scalable and can accommodate many concurrent connections. Its good replication mechanisms ensure it can scale with the application's needs. As open-source software, PostgreSQL is a cost-effective choice

		compared to other proprietary database systems. Moreover, its high compliance with SQL standards eases the learning curve for developers and ensures a smoother transition if moving from another SQL-based system. Lastly, PostgreSQL has a strong community, extensive documentation, promising support, and continuous system improvement. We also considered other databases like MySQL and MongoDB. MySQL, like PostgreSQL, is a relational database, but it has less extensive feature support and doesn't scale as efficiently for read-heavy workloads. MongoDB, a NoSQL database, is well-suited for handling unstructured data and scales horizontally very well. Still, it doesn't match PostgreSQL's ACID compliance, and the nature of our data doesn't require the flexibility that MongoDB offers. Hence, given the requirements of the common API and the advantages PostgreSQL brings, we've chosen it as our data store.
6	**Decision**	Use EF Core as our ORM tool in the common API.
	Justification	The choice to use Entity Framework (EF) Core in our API stems from various considerations. EF Core, an Object-Relational Mapping (ORM) tool, simplifies data manipulation by translating our object-oriented code into SQL queries for the database, freeing developers from writing complex SQL code and enhancing productivity. As a feature-rich ORM, EF Core supports various database functions, including transactions, migrations, batching, and complex querying. It also supports both code-first and database-first approaches, providing flexibility in development. EF Core's LINQ support enables developers to write queries in C#, making code easier to read, write, and maintain. The integration of EF Core with .NET Core makes it an excellent choice for a .NET Core API. It can fully utilize the performance, cross-platform support, and modern functionality provided by .NET Core. Moreover, EF Core supports various databases, including PostgreSQL, ensuring seamless integration with our chosen database. Its support for database migrations facilitates easy database schema changes over time, a feature beneficial for an evolving project like ours. Lastly, EF Core has an active community. It is backed by Microsoft, assuring regular updates, bug fixes, and extensive

		documentation, which eases the learning curve and encourages best practices. Other ORMs, such as Dapper and NHibernate, were also considered. Dapper is lightweight and performs faster but lacks many features of a full-fledged ORM like EF Core. NHibernate is powerful and flexible, but its steep learning curve and less active community made EF Core a more favorable choice for our needs.
7	**Decision**	Adopt the CQRS pattern in the common API.
	Justification	The decision to adopt the Command Query Responsibility Segregation (CQRS) pattern in our API stems from its unique capability to address the system's scalability, performance, and complexity requirements. CQRS allows us to handle read and write operations separately, aligning with the nature of our system where we expect a high volume of read operations as customers view vending machine details and product information. A smaller, but equally crucial, number of write operations as they register, log in, or make purchases. Using CQRS, we can optimize the read side for performance and the write side for data consistency, allowing each to scale independently based on the system's requirements. This becomes especially important in peak times or response to promotional events when the load on the system might significantly increase. CQRS allows us to add resources to whichever side is most stressed, helping maintain overall system performance. In addition, CQRS encourages a cleaner, more maintainable architecture. It separates reading and writing concerns into distinct models, reducing complexity and making it easier to understand and modify the system's parts independently. This benefits development and maintenance, as changes to the command model do not affect the query model and vice versa, reducing the risk of unintended side effects. We considered traditional CRUD operations, but the inability to independently scale and optimize read and write operations and the potential for growing complexity as our system evolves made CQRS a more compelling choice for our API.
8	**Decision**	Use the Geohash algorithm to find neighbors of a given location.
	Justification	The decision to use Geohash in our Vendomatic API is driven by its efficient spatial indexing, simplicity, scalability, and its ability to facilitate proximity-based searches. This makes it a powerful tool

		for quickly locating nearby vending machines without the need for complex calculations. Alternatives like direct distance calculation or k-NN algorithm were considered; however, they lack the simplicity and the efficient spatial indexing provided by Geohash. Direct distance calculation is computationally heavy, especially with many vending machines. While k-NN can be used for similar purposes, it typically requires a more complex setup and fine-tuning for optimal performance.

6. Architectural Mechanisms

6.1. Architectural Mechanism 1: Publish-Subscribe Style

A message queue (RabbitMQ) will be used to relay messages from vending machines to subscribers to manage system state which yields a highly efficient one-way distribution of information with very low-coupling of components.

6.2. Architectural Mechanism 2: Availability

Monitoring the system to detect faults and rollback to revert the system to a previously known good state to recover from faults will be used as availability tactics.

6.3. Architectural Mechanism 3: Modifiability

Reducing size of each module, increasing cohesion and reducing coupling will be used as modifiability tactics.

6.4. Architectural Mechanism 4: Performance

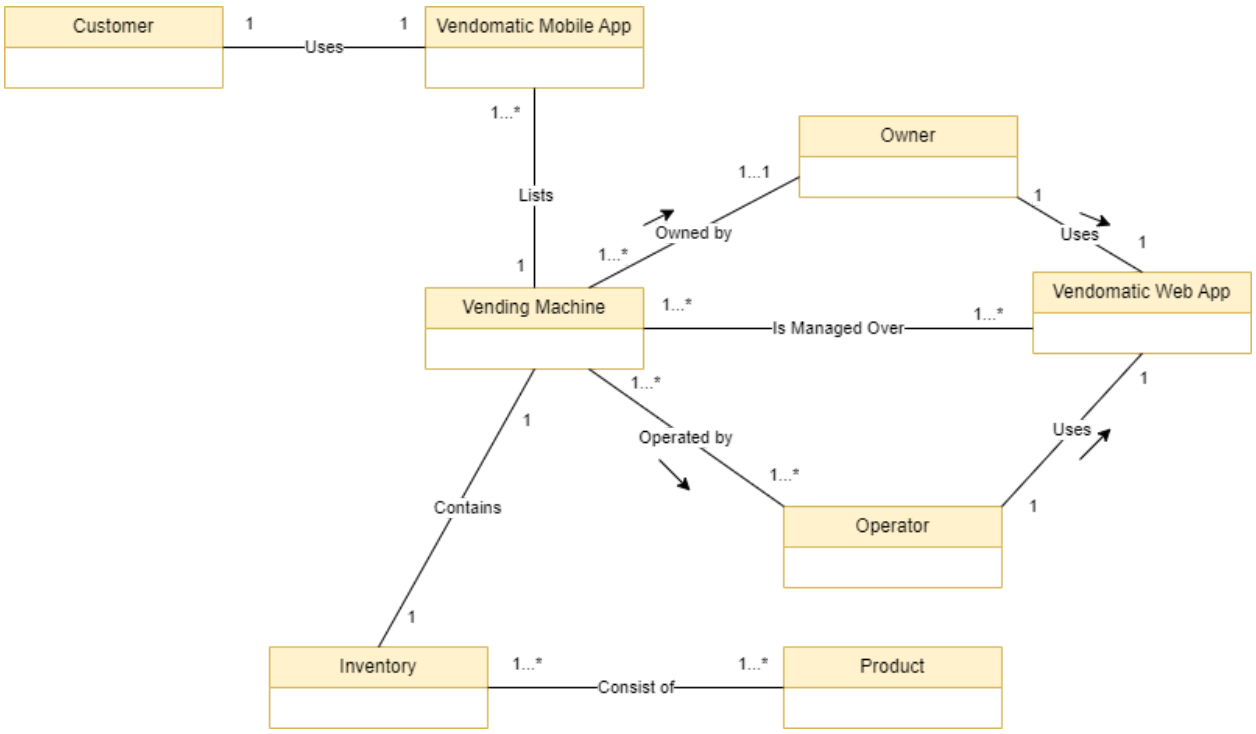
Limiting event responses and bounding execution times will be used as performance tactics.

6.5. Architectural Mechanism 5: Security

Actor authentication and authorization will be used to resist possible attacks. Also, limiting access to the system if there are repeated failed attempts to access will be used as a security tactic.

7. Key Abstractions

7.1. **Vending Machine:** Represents a physical vending machine in the system.
7.2. **Product:** Represents a product available in a vending machine.
7.3. **User:** Represents a customer, vending machine owner, or operator in the system.
7.4. **Maintenance Task:** Represents a maintenance activity assigned to an operator.
7.5. **Rating:** Represents a user's rating for a vending machine or product.



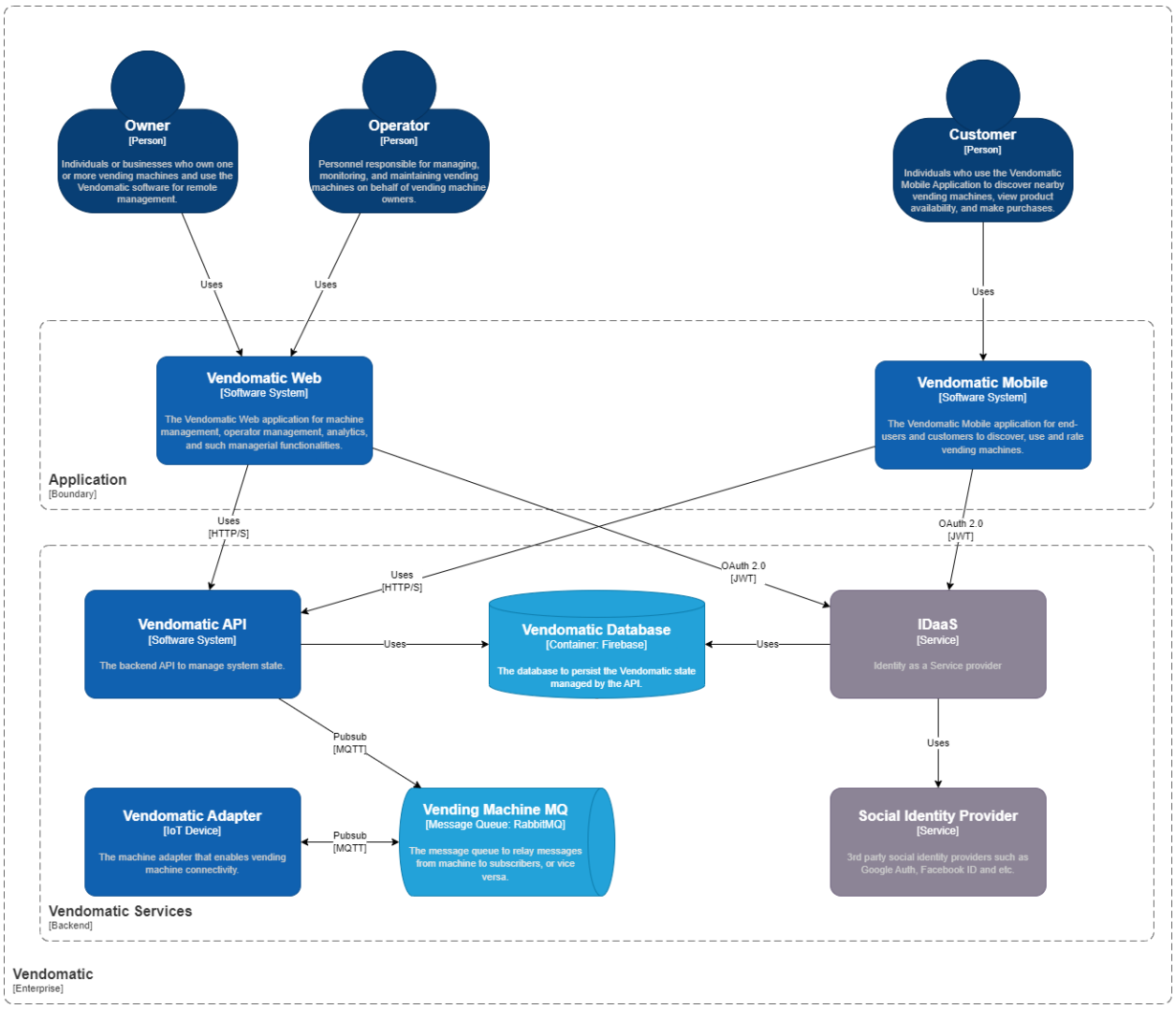
8. Architectural Framework

A layered architecture will be used, consisting of the following layers:

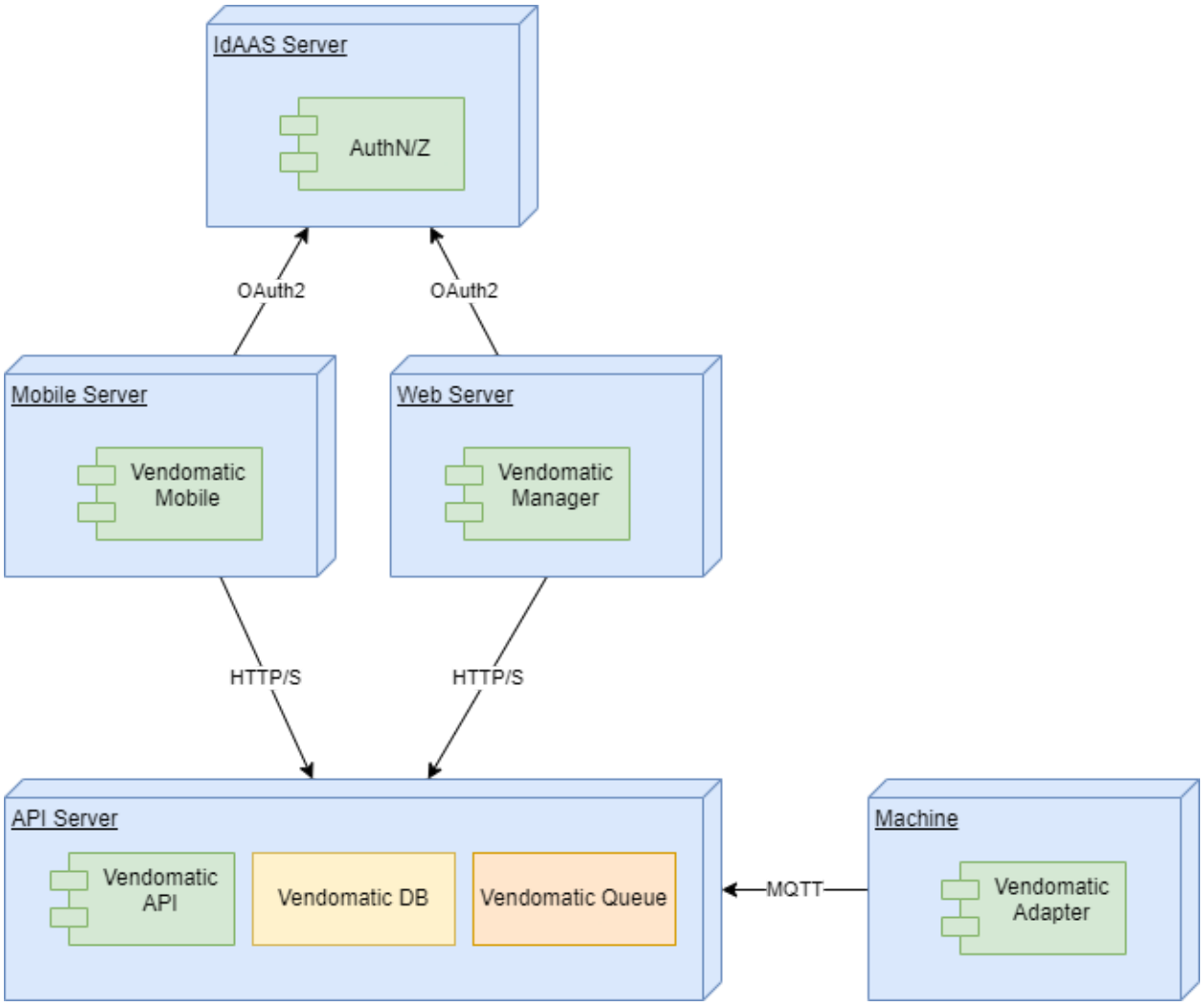
- **Presentation:** Web Application (for owners/operators) and Mobile Application (for customers).
- **Application:** Backend services responsible for business logic and coordination between other layers.
- **Domain:** Core domain model and business logic.
- **Data Access:** Responsible for storage and retrieval of data, including database access and caching.
- **Integration:** Facilitates communication with external systems, such as payment processors and map services.

9. Architectural Views

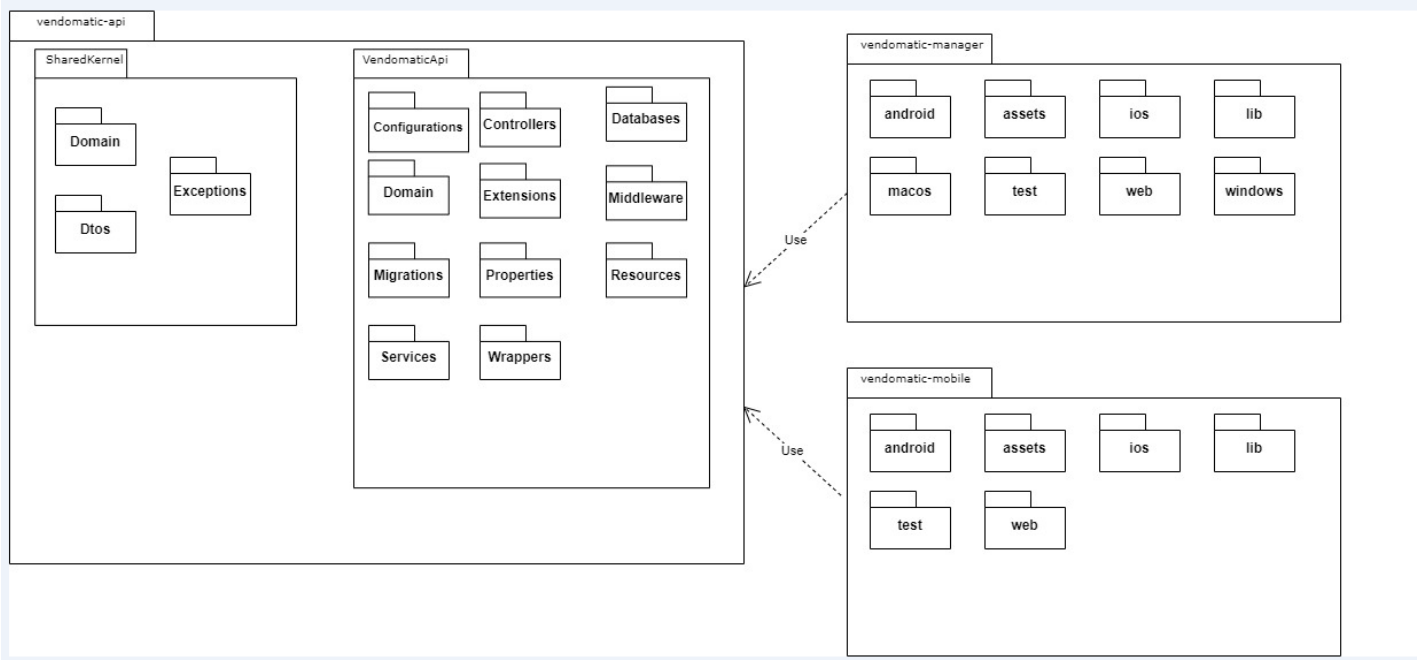
9.1. Context



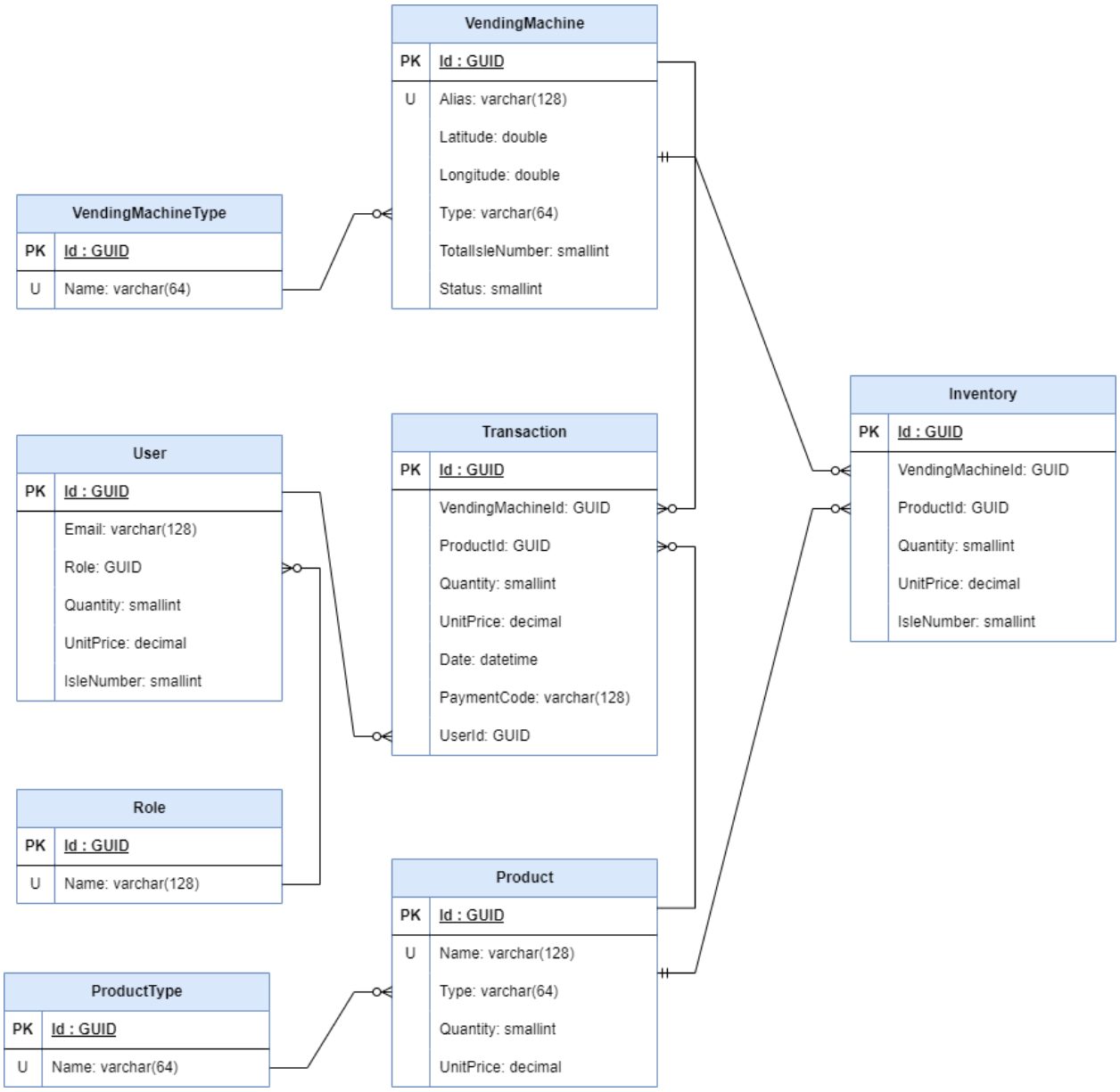
9.2. Physical View- Deployment Diagram



9.3. Logical View- Package Diagram



9.4. Data View


VendingMachine
PK Id : GUID
U Alias: varchar(128)
Latitude: double
Longitude: double
Type: varchar(64)
TotalIsleNumber: smallint
Status: smallint
VendingMachineType
PK Id : GUID
U Name: varchar(64)
Inventory
PK Id : GUID
VendingMachineId: GUID
ProductId: GUID
Quantity: smallint
UnitPrice: decimal
IsleNumber: smallint
Transaction
PK Id : GUID
VendingMachineId: GUID
ProductId: GUID
Quantity: smallint
UnitPrice: decimal
Date: datetime
PaymentCode: varchar(128)
UserId: GUID
User
PK Id : GUID
Email: varchar(128)
Role: GUID
Quantity: smallint
UnitPrice: decimal
IsleNumber: smallint
Role
PK Id : GUID
U Name: varchar(128)
Product
PK Id : GUID
U Name: varchar(128)
Type: varchar(64)
Quantity: smallint
UnitPrice: decimal
ProductType
PK Id : GUID
U Name: varchar(64)