ShopSmart
Architecture Notebook

1. Purpose

The purpose of this document is to offer a comprehensive architectural overview of the ShopSmart e-commerce application. It delves into the core philosophy, design decisions, constraints, and significant elements shaping the system's architecture. Through various architectural views, it aims to provide stakeholders with a clear understanding of the system's structure and behavior.

The document follows the 4+1 view model as the reference model for this document. With this way, this document presents both static and dynamic aspects of the system, catering to the needs of diverse stakeholders such as developers, designers, project managers, and system administrators. It elucidates the rationale behind key architectural decisions, including the integration of a product rating system and recommendation engine, highlighting their role in enhancing user experience and driving business success.

Ultimately, this Architecture Notebook serves as a guiding resource for the design and implementation of ShopSmart, ensuring alignment with its overarching goals while promoting scalability, reliability, and maintainability. On the other hand, all of the required diagrams and their descriptions are available in this document.

2. Architectural goals and philosophy

The architecture of ShopSmart is built upon the principles of adaptability, maintainability, and efficiency, with a keen focus on critical issues like hardware isolation and performance optimization. It seeks to foster collaborative development while remaining responsive to evolving technological and user needs. Core objectives include ensuring the system's maintainability throughout its lifespan, enabling seamless adaptation to changing requirements, and allowing data to evolve independently of its representations. Key considerations include isolating hardware dependencies and ensuring efficient performance under challenging conditions like high traffic volumes or intermittent network connectivity. In summary, the architecture serves as a robust foundation for the ShopSmart e-commerce platform, equipped to address both current and future challenges in the dynamic digital landscape.

Additionally, the Model-View-Controller (MVC) architecture allows for seamless collaboration among multiple developers, enabling concurrent work on the model, controller, and views. With MVC, related actions within the controller are logically grouped together, enhancing code organization and maintainability. Furthermore, views associated with a specific model are grouped together, promoting consistency and ease of management within the application's structure.

3. Assumptions and dependencies

The architectural decisions for ShopSmart are influenced by several key assumptions and dependencies crucial to project success. These assumptions and dependencies serve as guiding principles, ensuring that architectural decisions align with project objectives and constraints, ultimately leading to the successful development and deployment of the ShopSmart e-commerce application. These assumptions and dependencies can be summarized as below:

3.1 - Technological Flexibility with Integration Transparency:

• **Assumptions**: Technologies used in different system components can vary depending on the specific requirements and context of each component.
• **Dependencies**: While technological choices may vary, integration among these components must be seamless and transparent to end-users. This ensures a consistent user experience across different parts of the system.

3.2 - Technology Approval Process:

• **Assumptions**: The selection of technologies for each component involves collective decision-making.
• **Dependencies**: All technologies used must be discussed, reviewed, and approved within the team. Maintaining a manageable variety of technologies is preferred to streamline development and support processes.

3.3 - Collaborative Development:

• **Assumptions**: The project's scope and the geographical distribution of development teams necessitate a collaborative approach to the system's development and integration.
• **Dependencies**: There will be regular and periodic meetings between members of different teams to facilitate the integration of various components. This collaboration is critical to aligning team efforts and ensuring that integration points meet the design specifications.

3.4 - Skill and Experience of the Team:

• **Assumptions**: The development team has varying levels of skill and experience with the technologies and methodologies employed in the project.
• **Dependencies**: Training and continuous learning are essential for ensuring that all team members are competent in the technologies used and the architectural patterns adopted. This dependency impacts project timelines and quality.

3.5 - Availability of Resources:

• **Assumptions**: Essential resources, such as development tools, software licenses, and server capacities, are assumed to be adequately available to meet project demands.
• **Dependencies**: The availability of these resources is crucial for timely development and deployment. Any delays in procuring or accessing these resources could impact project milestones.

3.6 - Compliance and Security Standards:

• **Assumptions**: The system must comply with relevant data protection and security regulations.
• **Dependencies**: This compliance dictates certain architectural choices, such as the use of specific secure coding practices, encryption methods, and audit mechanisms. The architecture must support these requirements inherently to avoid costly reworks.

4. Architecturally significant requirements

• **Usability**: The system shall support advanced usability protocols to ensure that users can navigate the website intuitively and efficiently. This requires a responsive design and logical user interface that adheres to established UX principles.

• **Reliability**: The system must be robust, capable of handling high traffic volumes and maintaining high availability. This involves implementing fault-tolerant mechanisms and ensuring that critical components can handle unexpected loads and recover from errors without significant downtime.

• **Performance**: Users should experience smooth operations within the system, which necessitates high performance. This includes optimizing backend operations, efficient data retrieval and storage solutions, and minimal response times for user interactions.

• **Maintainability and Upgradability**: The architecture must facilitate easy maintenance and future upgrades. This requires the use of modular components, adherence to coding standards, and implementation of a clear code documentation strategy to allow for easy changes and additions.

• **Security**: Critical security measures must be integrated into the system, especially in areas involving user data and financial transactions. This includes securing user credentials, using HTTPS for data transmission, and leveraging security features provided by frameworks like Spring for secure authentication and authorization processes.

Each of these requirements not only influences specific architectural choices but also dictates how the system will be structured and developed to meet these needs. For example, the decision to use a modular architecture directly supports the requirements for maintainability and security, allowing different parts of the system to be updated or replaced independently without affecting the overall system integrity.

5. Decisions, constraints, and justifications

5.1.1 - Use of MVC Architecture with Spring Boot

Decision: All development must adhere to the MVC model integrated with Spring Boot.
Justification: This decision ensures that the application structure is modular, which facilitates independent development and testing of components. Using Spring Boot enhances security and reduces boilerplate code, speeding up development. The modular approach allows for easier updates and maintenance, aligning with our goal for a secure and modern application architecture.

5.1.2 - Implementation of Fault-Tolerant Systems

Constraint: The system architecture must include redundant components to enable continuous operation, even during component updates or failures.
Justification: By designing a fault-tolerant system with redundant components, we ensure high availability and service continuity. This is crucial for maintaining user trust and satisfaction, as it minimizes downtime and service disruptions, essential for systems requiring high availability like financial or critical services applications.

5.1.3 - Emphasis on Maintainability over Performance

Constraint: The system shall prioritize maintainability; thus, developers should design using fine-grained, self-contained components. However, this may impact performance as it could lead to larger overall component footprints.
Justification: While performance is important, the longevity and adaptability of the software are prioritized to accommodate future changes and enhancements without extensive rework. This decision supports our goal of building a "non-aging" software platform. Developers are encouraged to optimize the flow within the existing constraints to ensure that the system remains responsive and efficient.

5.1.4 - Continuous Integration and Regression Testing

Constraint: The development process must include continuous integration (CI) and regular regression testing.
Justification: Continuous testing is essential to maintain the reliability of the system. It allows developers to detect and resolve issues early, ensuring that new changes integrate smoothly without disrupting existing functionalities. This approach reduces the risk of bugs and improves the overall quality of the software.

5.2 - List of DOs and DON'Ts

• **DO** adhere strictly to the MVC model and use Spring Boot for all development.
• **DO** design with redundancy and fault tolerance in mind to ensure system availability.
• **DO** focus on creating maintainable, modular components even if it might initially affect performance.
• **DO** engage in continuous integration and frequent regression testing to maintain system integrity.
• **DON'T** compromise on security protocols or bypass established architectural patterns for expedience.
• **DON'T** neglect the maintainability of the system for performance gains without thorough impact analysis.

6. Architectural Mechanisms

6.1 - Model

The Model is the central component of the pattern. It directly manages the data, logic, and rules of the application. In our architecture, the Model represents the knowledge of the domain, the data structure, and the business rules. It is responsible for responding to requests for information about its state (usually from the view), and responding to instructions to change state (usually from the controller).

Attributes:
• **Business Logic**: Handles calculations, data manipulation, and decision-making processes.
• **Database Integration**: Interacts directly with the database to retrieve and store data.

Functionality:
• **State Management**: Maintains and updates the application's state based on user interaction or other inputs.
• **Change Notification**: Notifies the View component of any changes in data to refresh the user interface.

6.2 - View

The View component is used for all the UI logic of the application, rendering data provided by the Model in a format suitable for interaction, typically user interfaces. The View observes the Model and updates the visual representation of the data when it changes. It generates output representations of data, providing a visual interface for the user, but does not perform any processing of the data itself.

Attributes:
• **Dynamic Display**: Capable of rendering changes in data in real time.
• **User Interaction**: Facilitates interaction with the user through various forms of input and controls.

Functionality:
• **Render Data**: Displays data provided by the Model.
• **User Event Handling**: Captures user actions and sends them to the Controller.
• **Update Requests**: Requests updates from the Model to refresh displayed data.

6.3 - Controller

The Controller acts as an interface between the Model and the View components. It listens to the user inputs, typically through actions or interactions with the View, and processes these inputs by making calls to Model objects to retrieve data or to update the View. The Controller interprets the inputs, converting them into commands for the Model or View.

Attributes:
• **HTTP Request Handling**: Manages incoming and outgoing HTTP requests.
• **Application Logic**: Implements the specific logic necessary to handle user requests.
• **Data Validation**: Ensures that incoming data meets validation rules before processing.

Functionality:
• **Request Handling**: Processes user inputs and maps them to Model updates or View updates.
• **Output Generation**: Selects the appropriate View and provides it with the necessary data from the Model to render the output.

7. Key abstractions

7.1 - Critical Abstractions

• **Customer:** Represents the end-users of the online shopping system. This abstraction is responsible for capturing all the activities and characteristics of a user, such as browsing items, placing orders, and managing personal accounts.

• **Merchant:** Defines the vendors on the platform who list their products for sale. This abstraction includes functionalities related to product management, sales analytics, and inventory control, enabling sellers to effectively supply and manage their product offerings.
• **Admin:** Represents system administrators who oversee and manage the operation of the online shopping platform. This abstraction is crucial for tasks such as user and seller management, system monitoring, and enforcing compliance with policies and regulations.
• **Items:** Represents the products listed on the platform. This abstraction is central to the shopping system, encompassing the properties of each item such as price, description, stock levels, and category.
• **Community Moderator:** Represents personnel who are essential in managing and overseeing the community elements of the online shopping platform. This role is fundamental for moderating user reviews, comments, and forums to promote a respectful and constructive environment. Community Moderators are tasked with enforcing community guidelines, resolving disputes, and ensuring that interactions within the platform remain positive and productive.

7.2 - Other Abstractions

• **Interfaces:** Describes the various user interfaces that facilitate interaction between the system and its users—customers, sellers, and administrators. This abstraction is crucial for delivering a seamless user experience across different platform functionalities.
• **Database:** Acts as the central repository for all data associated with the online shopping system. This includes storing user profiles, product details, transaction records, and other pertinent data, ensuring data integrity and accessibility.

8. Layers or architectural framework

Our system employs the Model-View-Controller (MVC) architectural pattern, a well-established design framework that separates an application into three interconnected components. This separation helps manage complexity, promotes organized code development, and facilitates scalability and maintenance.

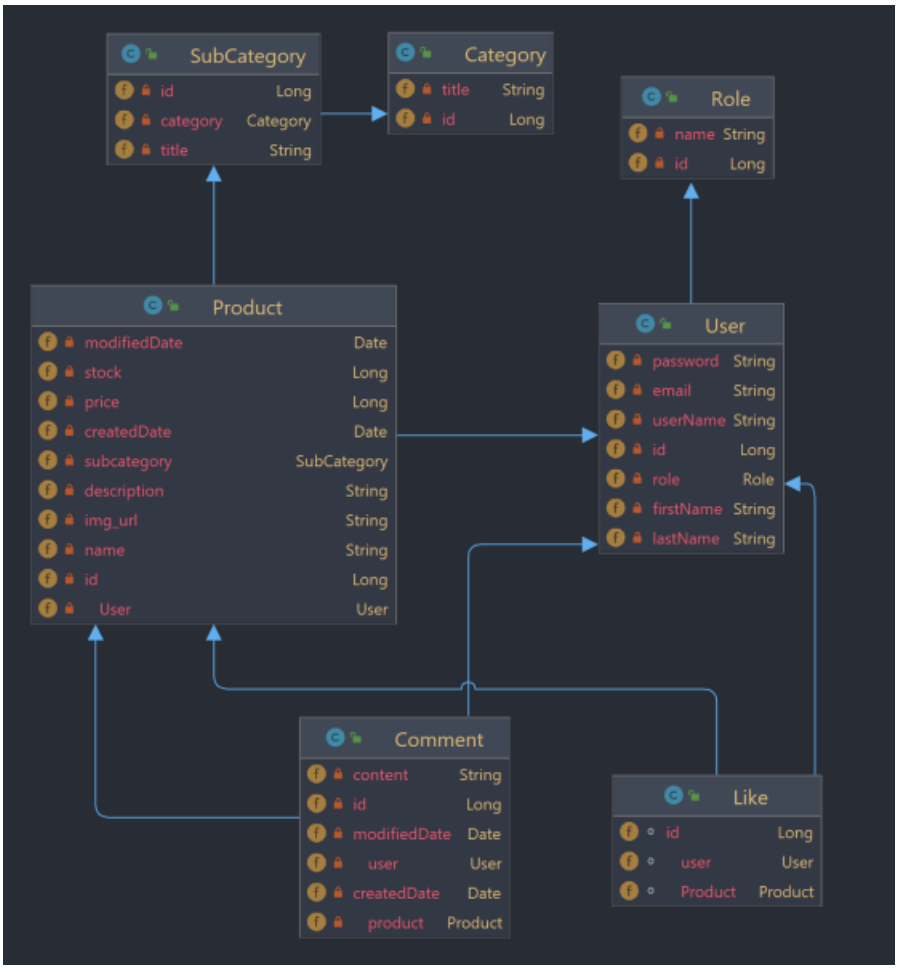
Consistency: The MVC architecture provides a consistent approach to organizing application programming. Each component is designed to handle specific development aspects of an application. MVC separates the business logic from the user interface, which results in an application where the backend logic and frontend UI can be developed independently.

Uniformity: Implementing MVC enables uniformity across the application by ensuring that each component only handles specific tasks. This separation of concerns ensures that the system is organized according to a clear structure, where:
• The Model manages the fundamental behaviors and data of the application, reacts to instructions, changes state, and notifies the View of changes.
• The View displays the data (the Model) in specific formats and interfaces, responding to user inputs by notifying the Controller.
• The Controller handles user input, interacts with the Model, and selects a View for response.

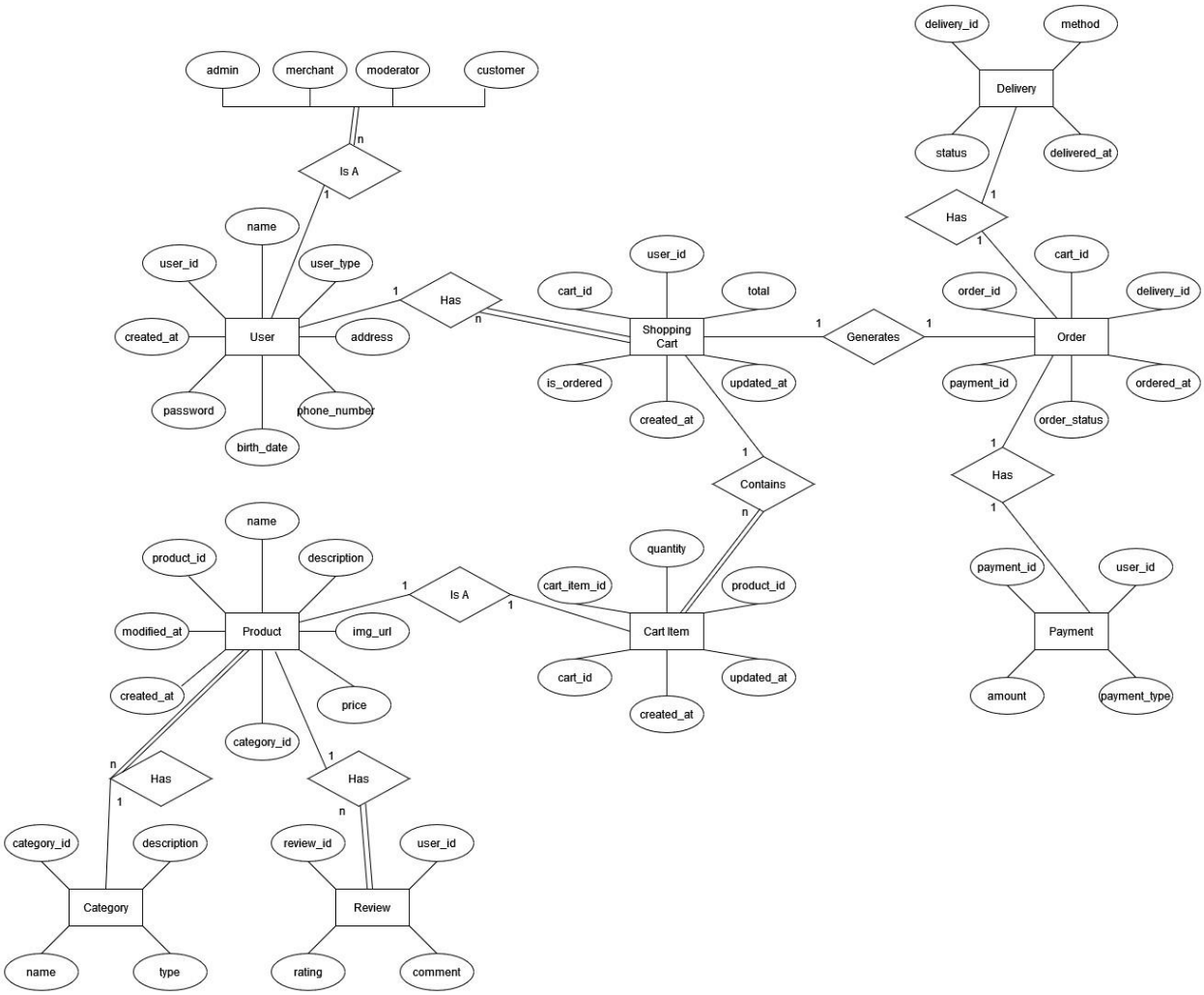
9. Architectural views

9.1 - Logical View

• Class Diagram

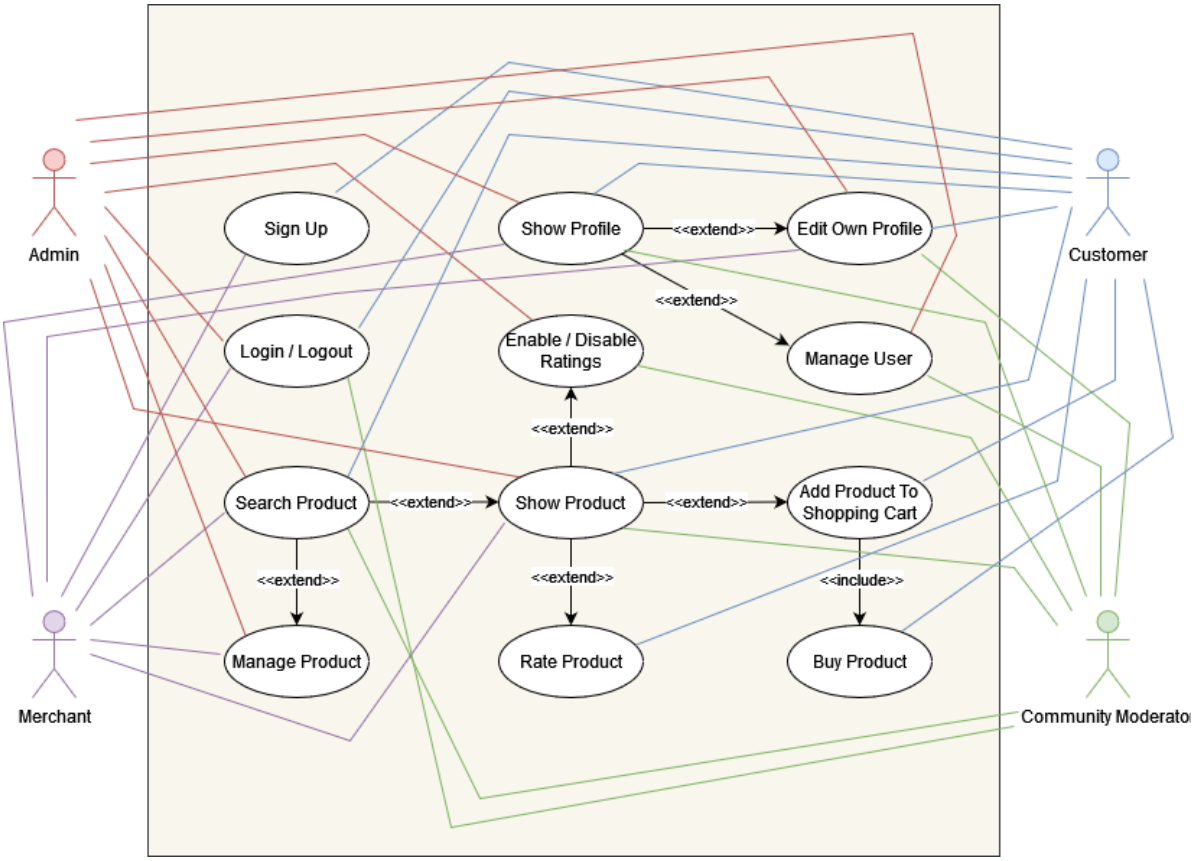


• E/R Diagram


admin
merchant
moderator
customer
Is A
name
user_id
user_type
created_at
User
address
password
phone_number
birth_date
Has
cart_id
user_id
total
Shopping Cart
is_ordered
updated_at
created_at
Generates
delivery_id
method
Delivery
status
delivered_at
Has
cart_id
order_id
delivery_id
Order
payment_id
ordered_at
order_status
Has
payment_id
user_id
Payment
amount
payment_type
Contains
quantity
cart_item_id
product_id
Cart Item
cart_id
updated_at
created_at
name
product_id
description
modified_at
Product
img_url
created_at
price
category_id
Is A
Has
category_id
description
Category
name
type
Has
review_id
user_id
Review
rating
comment

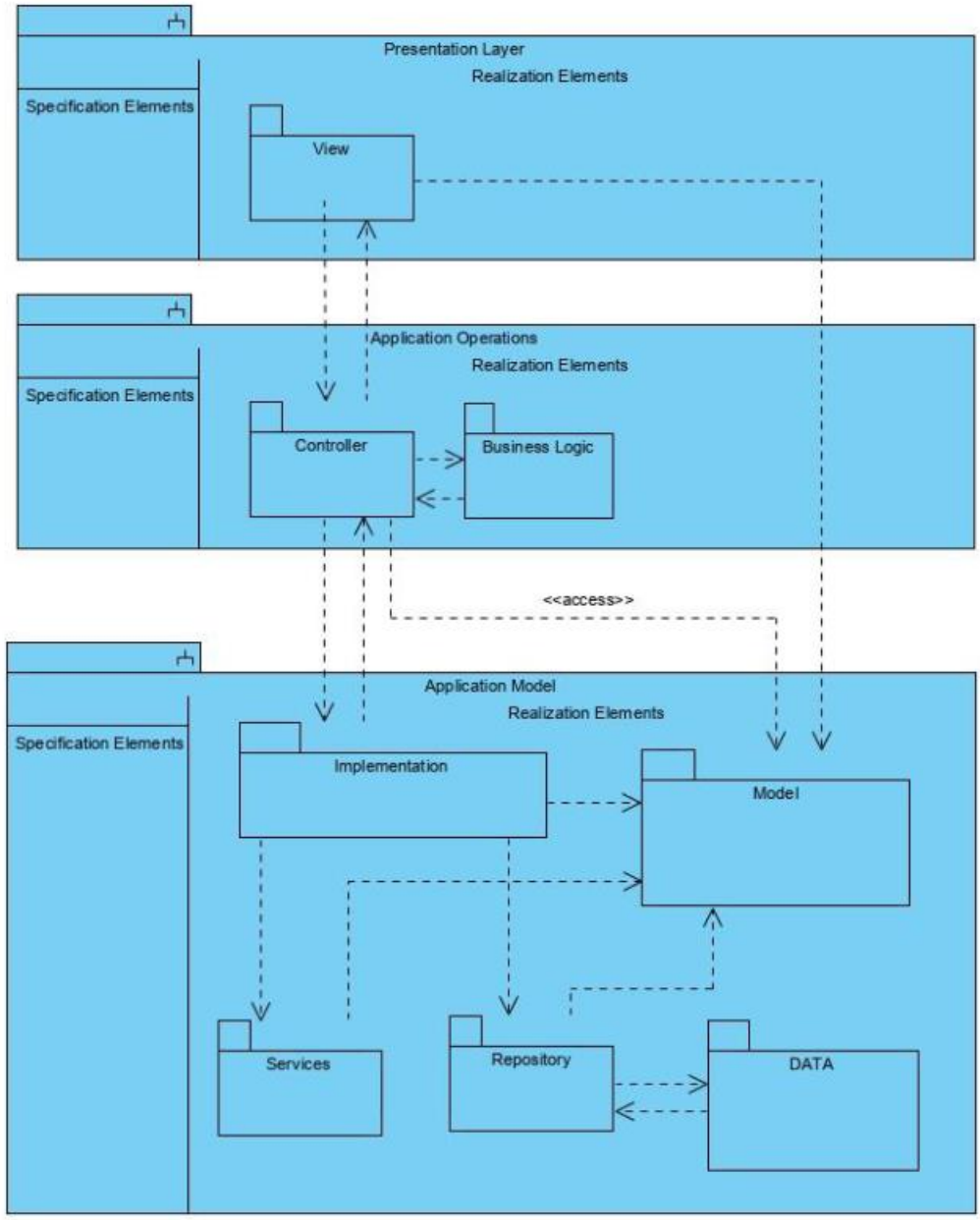
9.2 - Use Case View

• Use Case Diagram

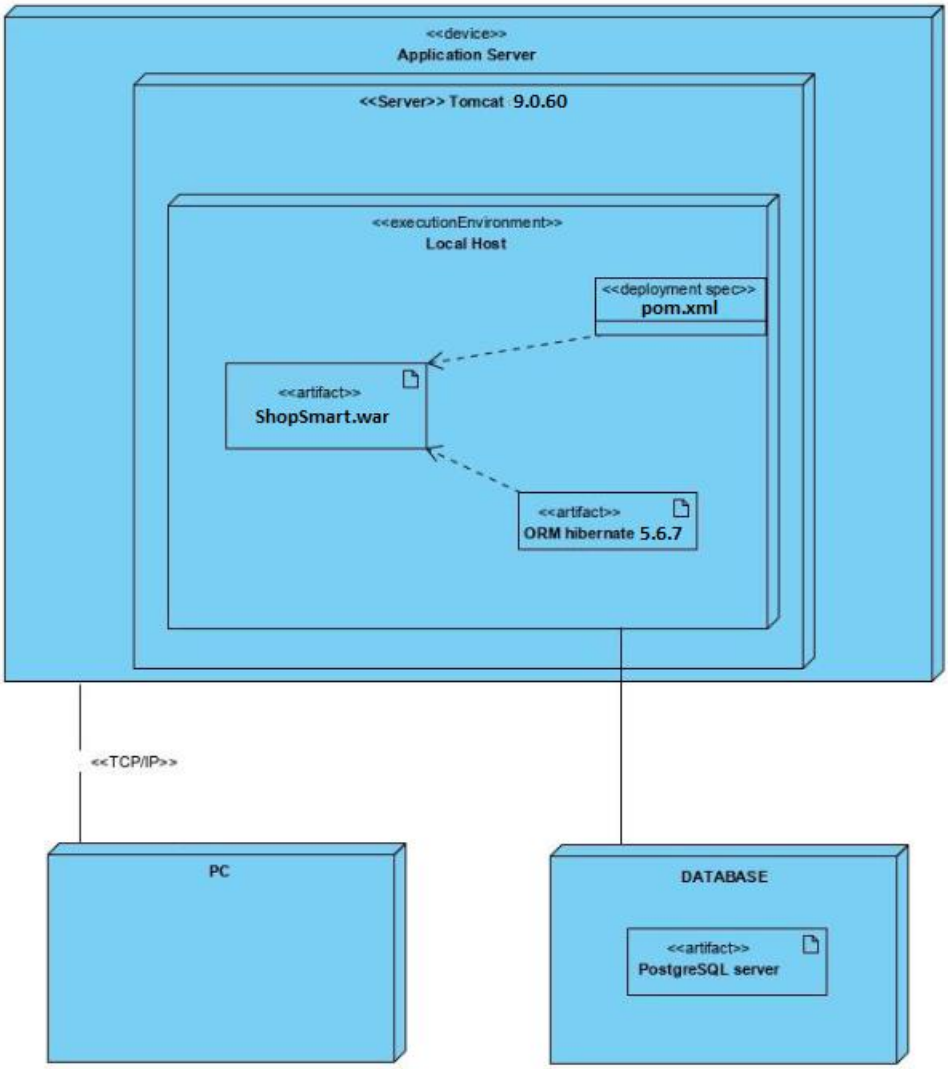


9.3 - Operational View

• Package Diagram



• **Deployment Diagram**



• **Component Diagram (Does Not Exist!)**