High-Level Architecture Design and Kruchten's 4+1 Views

Course Instructor

Dr. Atif Jilani

Submitted By

Aamina Bokhari_22I-2474

Ahmad Javed_21I-1108

Umama Saif_22I-2558

Date

May 13, 2024

Spring 2024

Department of Software Engineering

FAST – National University of Computer & Emerging Sciences

System Architecture:

We have chosen the **MVC (Model-View-Controller)** architecture for the BloodFlowPro system due to its proven track record of promoting modular, scalable, and maintainable software design.
The adoption of the MVC (Model-View-Controller) architecture for the BloodFlowPro blood bank management system is driven by several key factors:

1. **Separation of Concerns:**
    - MVC promotes a clear separation of concerns, dividing the application into three interconnected components: Model, View, and Controller. This separation enhances modularity, maintainability, and scalability, allowing developers to focus on specific aspects of the application without interfering with others.

2. **Scalability and Maintainability:**
    - By organizing the application into distinct components, MVC facilitates scalability and maintainability. Each component can be developed, tested, and maintained independently, enabling parallel development efforts and reducing the risk of introducing unintended side effects when making changes.

3. **Enhanced Testability:**
    - MVC architecture inherently supports unit testing and test-driven development (TDD). With clear boundaries between the Model, View, and Controller components, it becomes easier to write isolated tests for each component, ensuring the reliability and correctness of the system.

4. **Flexibility and Reusability:**
    - MVC promotes code reusability and flexibility by allowing interchangeable components. For instance, the same Model layer can be utilized across multiple Views or Controllers, promoting consistency and reducing development effort.

5. **Support for Multiple User Interfaces:**
    - MVC architecture enables the development of multiple user interfaces (UIs) using the same underlying business logic (Model). This flexibility is crucial for BloodFlowPro, as it may require web-based interfaces for administrators, mobile applications for donors, and specialized interfaces for healthcare providers.

6. **Enhanced Collaboration:**
    - MVC's clear separation of concerns facilitates collaboration among development teams. With well-defined interfaces between components, teams can work concurrently on different parts of the application without stepping on each other's toes, leading to improved productivity and reduced development time.

7. **Adaptability to Change:**

- BloodFlowPro is a dynamic system that may undergo frequent updates and enhancements. MVC's modular architecture makes it easier to accommodate changes without affecting the entire system. For example, changes to the View layer (UI) can be implemented without modifying the underlying business logic (Model) or application flow (Controller).

In summary, the MVC architecture provides a robust foundation for developing the BloodFlowPro blood bank management system, offering benefits such as separation of concerns, scalability, maintainability, testability, flexibility, and adaptability to change. By leveraging MVC principles, BloodFlowPro can achieve a well-structured, maintainable, and extensible solution that meets the evolving needs of blood bank management effectively.

Architecture Diagram:

Explanation:

- **Presentation Layer:**

- JavaFX UI (View): This block encompasses all the visual elements of the application, such as windows, forms, buttons, tables, and other UI components. It is responsible for displaying information to the user and receiving input.
- JavaFX Controllers: These classes act as intermediaries between the UI and the underlying application logic. They handle user interactions (button clicks, form submissions), validate input, and update the UI based on data changes.

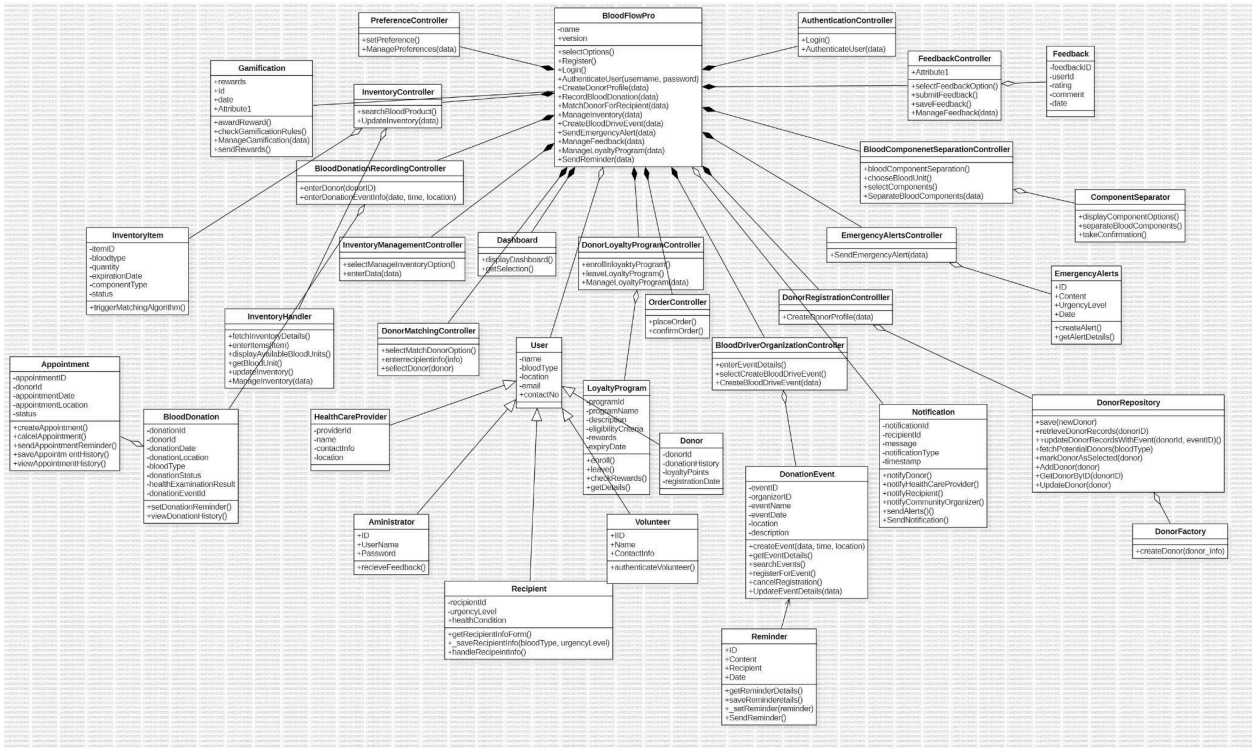
- **Application Layer:**

- Service Layer (Business Logic): This layer contains the core logic of the application. Services encapsulate business rules, validate data, orchestrate workflows, and interact with the DAO layer to retrieve or update data.
- DAO Layer (Data Access): The DAO (Data Access Object) layer provides an abstraction for interacting with the database. DAOs are responsible for executing database queries (CRUD operations) and mapping the results to domain objects (e.g., Donor, BloodInventory).

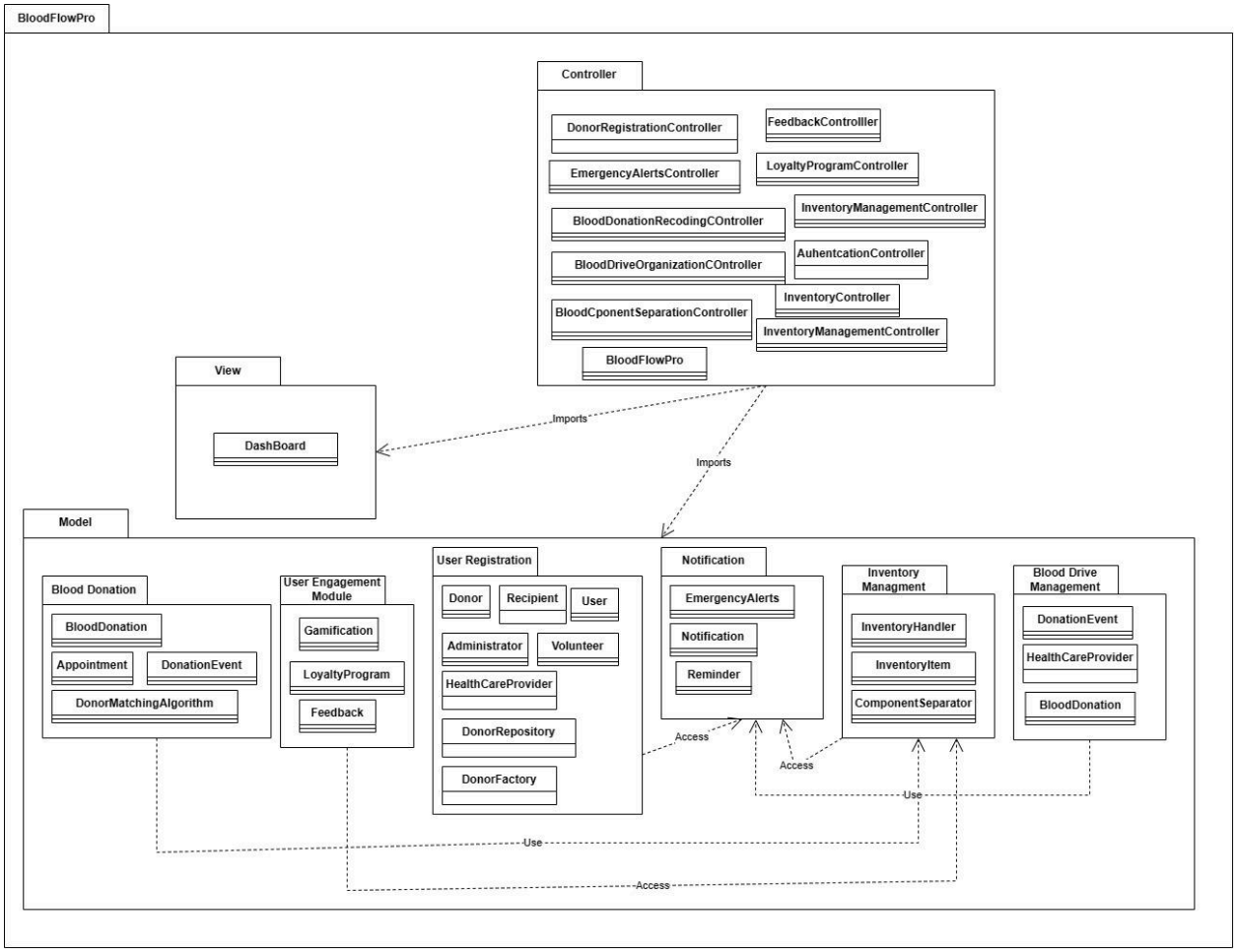
- **Data Layer:**

- Database: This is the persistent storage for the application's data. It could be a relational database like MySQL or PostgreSQL, or a NoSQL database, depending on your requirements.

Class Diagram:



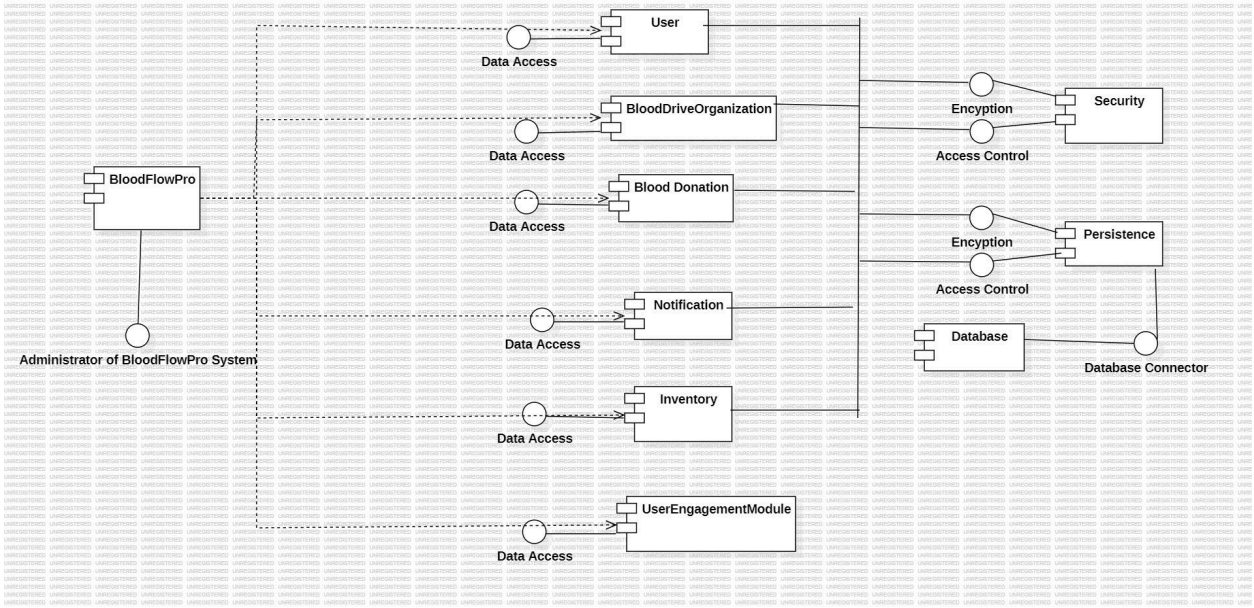
Package Diagram:



Explanation:

The image is a UML package diagram for a blood bank system. It's divided into three main packages: Controller, View, and Model. The Controller manages various aspects of the system, the View displays data, and the Model contains data related to blood donation and user management. The diagram also shows the interactions between these components.

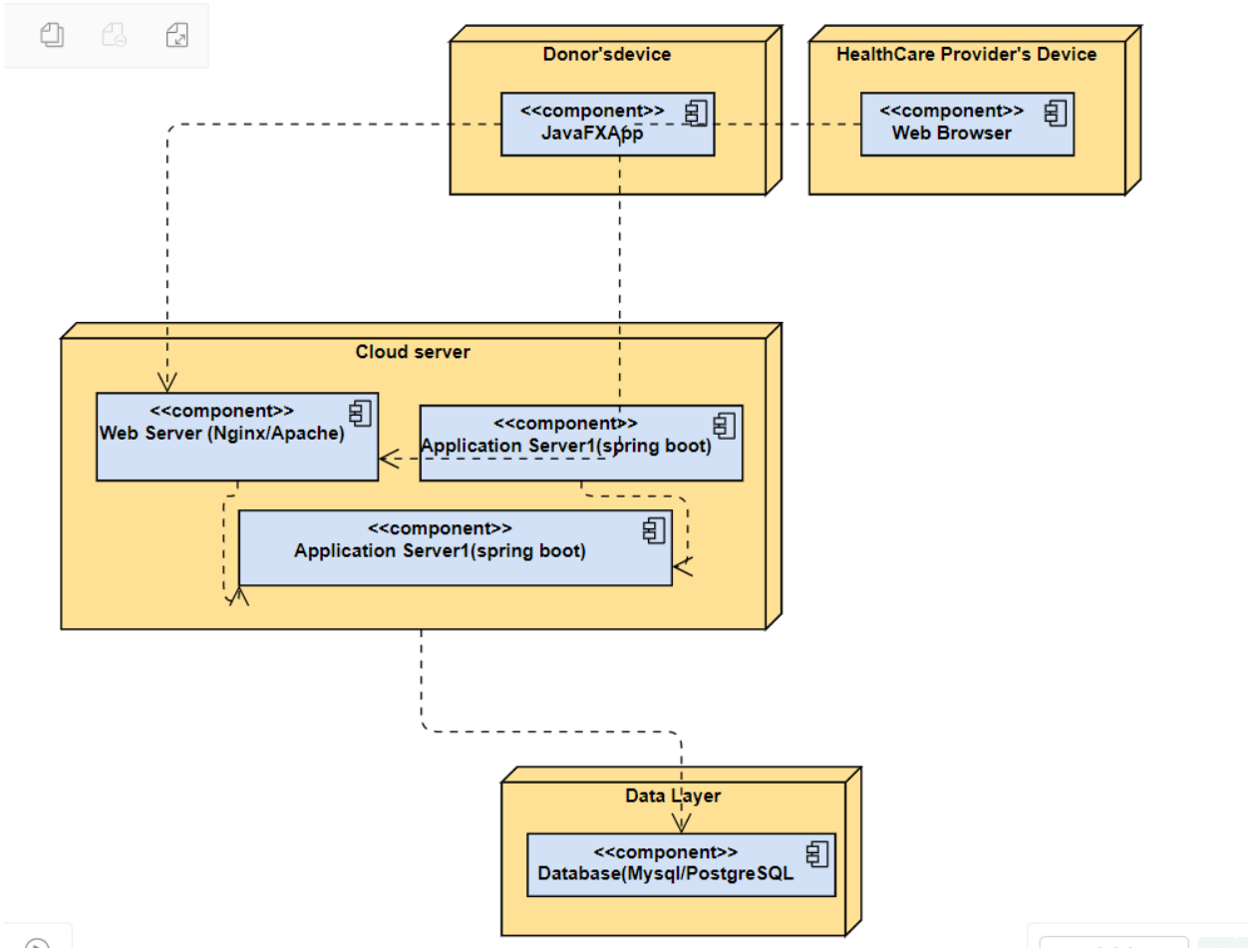
Component Diagram:



Explanation:

The component diagram provided illustrates the architecture of a blood donation system. The system is composed of several key components including “User,” “BloodDriveOrganization,” “Blood Donation,” “Notification,” “Inventory,” and “UserEngagementModule.” Each of these components is associated with a “Data Access” component, indicating a mechanism for retrieving and storing data.

Deployment Diagram:



Explanation:

1. **Donor's Device:**
    - The donor interacts with the system through a JavaFX application installed on their laptop or mobile device. This application provides the UI for registering, recording donations, viewing history, etc. (Use Cases 1, 2, 8, 9).
2. **Healthcare Provider's Device:**
    - Healthcare providers access the system via a web browser on their PC. They use the web interface to request blood, view inventory, and potentially access donor matching features (Use Cases 3, 4, 6).
3. **Cloud Server:**
    - **Web Server (Nginx/Apache):** Handles incoming requests from both the JavaFX application and the web browsers, routing them to the appropriate application server.

    - **Application Server 1 & 2 (Spring Boot):** Multiple application servers can be deployed for load balancing and redundancy. Each server runs the Java Spring Boot application, which contains the business logic, services, DAOs, and controllers.
    - **Load Balancer (Optional):** Distributes incoming traffic across the multiple application servers to ensure optimal performance and availability.
4. **Database:**
    - The database (MySQL or PostgreSQL) stores all the persistent data for the system, including donor information, inventory levels, donation records, etc. Multiple database instances could be set up for replication and backup.
5. **Network (Internet):**
    - All communication between the client devices and the cloud server occurs over the internet.

Use Case Mapping:

- The JavaFX application on the donor's device communicates directly with the application servers to handle use cases related to donor registration, donation recording, and history tracking.
- Healthcare providers' browsers interact with the web server, which then forwards requests to the application servers to handle inventory management, donor matching, and blood requests.
- The application servers interact with the database to fetch and store data required for all use cases.