AnyChange
Architecture Notebook

1. Purpose

This document describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the system that shape the design and implementation.

2. Architectural goals and philosophy

The architecture of AnyChange has to be adaptable to integrate with many different external product selling platforms and has to be scalable to meet the performance criteria of a growing customer base. It should provide a secure way of communicating between the users and application software.

Scalability: The architecture must be scalable to handle the increasing customer base and growing transaction volumes. As the number of users and concurrent requests rises, the system should gracefully handle the load without compromising performance. The scalability solution should ensure that response times remain within acceptable limits and that the system can be easily expanded as the business expands.

Integrity: Data integrity is essential to maintain the accuracy and consistency of the system's data. The architecture should include mechanisms to validate and protect data from unauthorized modifications, corruption, or loss. This includes implementing appropriate data validation checks, employing data encryption and backup strategies, and ensuring the integrity of data transfers between components**.**

Maintainability: The architecture should be designed and implemented with maintainability in mind, facilitating efficient development and future enhancements. Consider the following factors:

Modularity: Design the system with clear modular boundaries, allowing for independent development, testing, and maintenance of different components. This promotes code reusability and makes it easier to understand and modify specific parts of the system.

Documentation: Provide comprehensive documentation that describes the architecture, design patterns, and component interactions. This documentation should also include guidelines and best practices for developers to follow, ensuring consistency and ease of maintenance.

Code Maintainability: Emphasize clean code practices, such as following coding conventions, writing self-explanatory code, and properly organizing the codebase. Employ automated testing, continuous integration, and code review processes to catch and address issues early, reducing technical debt and enhancing maintainability.

Version Control: Utilize a robust version control system (e.g., Git) to track changes, manage branches, and facilitate collaboration among developers. Use branching and tagging strategies to support parallel development efforts and enable easy rollback or deployment of specific versions.

Monitoring and Logging: Implement logging and monitoring mechanisms to capture system events, errors, and performance metrics. These tools provide valuable insights into the system's behavior, aiding in identifying and resolving maintenance issues promptly.

3. Assumptions and dependencies

Assumptions:

The following assumptions are made for the AnyChange architecture:

- Team Composition: The development team is composed of developers with varying levels of experience, including junior, intermediate, and senior developers. They possess the necessary skills to work with the chosen technologies and frameworks.
- Legacy Interfaces: AnyChange will be built from the ground up, without dependencies on legacy interfaces. This assumption allows for a clean and independent implementation of the system, avoiding any constraints or limitations imposed by existing interfaces.
- Web-Based Software: AnyChange is a web-based software product, designed to be accessed and used through web browsers. It assumes that users will interact with the system primarily through web-based interfaces.
- Availability of APIs: It is assumed that there won't be any publicly available APIs to pull price data for products. This assumption leads to the decision of using an external scripting platform for web scraping to collect price data from various sources. However, if any relevant APIs become available, the system should be able to integrate and utilize them accordingly.

Dependencies:

The following dependencies exist for the AnyChange architecture:

- Security: The system depends on ensuring the security of all data, including the database applications and clients. This includes:

  Data Encryption: Critical data such as credit card information and user passwords must be encrypted to protect against unauthorized access or attacks.

  Authentication: All actions performed on the user's account should be authenticated to prevent unauthorized access and malicious use.

  Communication Security: All communications between users and the AnyChange platform must be encrypted using HTTPS, ensuring secure transmission of sensitive data over public networks.

- Functional Uptime: The system depends on maintaining a high level of availability and functionality. It should be operational and reachable during its designated usage hours. This dependency includes:

  Software Updates: The system should remain available and operational during software updates or patches, minimizing downtime and ensuring a seamless user experience.

  Component Failures: In the event of a component failure or crash, the system should continue to operate as expected, with appropriate redundancy or failover mechanisms in place to ensure uninterrupted service.

- Scalability: The architecture depends on the ability to scale up to meet the increasing demands of a growing customer base and transaction volumes. This dependency includes:

  Load Handling: As the number of users and concurrent requests increases, the system should gracefully handle the load without compromising performance. Response times should remain within acceptable limits.

  Business Expansion: The architecture should support easy expansion as the business expands, allowing for the addition of new features, modules, or infrastructure components to accommodate growth.

- Concurrent Transactions: The system depends on efficient concurrent processing mechanisms to support a large number of simultaneous transactions. This dependency includes:

  Parallel Processing: The architecture should enable parallel processing and distributed services to handle multiple transactions concurrently, ensuring optimal performance and responsiveness.

4. Architecturally significant requirements

The following architecturally significant requirements have been identified for the AnyChange system:

1. Security: All data, database applications, and clients must be secure. This includes the encryption of critical data such as credit card information and user passwords to protect against unauthorized access.
2. Functional Uptime: The system should have a functional uptime of 166 hours per week, ensuring that it remains available and operational during its designated usage hours.
3. Scalability: The architecture needs to be scalable to handle the increasing customer base and growing transaction volumes. It should gracefully handle the load without compromising performance and response times.

4. Decisions, constraints, and justifications

Flexibility in Data Collection:
Decision: AnyChange will use an external scripting platform for web scraping to accommodate platforms without publicly available APIs.
Justification: This decision allows flexibility in collecting data from various platforms by leveraging custom scraping solutions for each source. It ensures that a wide range of products and sources can be monitored.
Alternative Approach: If publicly available APIs are available for all platforms, using those APIs directly could simplify the data collection process and reduce the complexity of maintaining custom scraping solutions.

Model-View-Controller (MVC) Architecture:
Decision: AnyChange will be built using the Model-View-Controller architectural pattern, supported by the Spring Boot framework.
Justification: MVC promotes a separation of concerns, enabling modularity and maintainability. The Spring Boot framework provides robust support for MVC, simplifying development and leveraging the team's existing experience.
Alternative Approach: Alternative architectural patterns like Microservices or Hexagonal Architecture could be considered depending on the complexity and specific requirements of the system. These patterns offer different trade-offs in terms of scalability and maintainability.

Java as the Development Language:
Decision: AnyChange will be developed using Java for cross-platform operation.
Justification: Java is a widely used programming language known for its portability, strong ecosystem, and extensive libraries. It enables cross-platform compatibility and offers a mature development environment.
Alternative Approach: Depending on specific requirements and the development team's expertise, alternative languages like Python, JavaScript, or C# could be considered. Each language has its own strengths and may be better suited for certain aspects of the system.

Python as the External Scripting Platform:
Decision: AnyChange will use Python as the external scripting platform for web scraping.
Justification: Python is known for its simplicity, ease of use, and rich libraries for web scraping. It allows for the efficient development of custom scraping solutions.
Alternative Approach: Depending on the team's expertise and the specific scraping requirements, other scripting languages like Ruby or Node.js could be considered. Each language has its own ecosystem and community support for web scraping.

Encryption and Security:
Decision: All communications between users and the AnyChange platform will be encrypted using HTTPS. Critical data such as credit card information and passwords will be encrypted.
Justification: Encryption ensures the confidentiality and integrity of sensitive data during transmission and storage, protecting user privacy and preventing unauthorized access.
Alternative Approach: Implementing additional security measures like two-factor authentication, token-based

authentication, or data anonymization techniques could enhance the overall security posture of the system.

Authentication and Authorization:
Decision: Any action on the user's account will be authenticated. Users will be authorized based on their type (normal user, seller user, and admin) to prevent unwanted actions.
Justification: Authentication ensures that only legitimate users can perform actions within the system, while authorization restricts access based on user roles to prevent unauthorized operations.
Alternative Approach: Implementing role-based access control (RBAC) or attribute-based access control (ABAC) mechanisms could provide more granular control over user permissions and enhance security.

Password Change Notifications:
Decision: Actions like changing passwords will require informing the user through a separate channel, such as email, to prevent malicious use.
Justification: Requiring an additional channel for sensitive actions adds an extra layer of security, reducing the risk of unauthorized changes and ensuring user awareness and consent.
Alternative Approach: Depending on the system requirements, alternative methods for notifying users about sensitive actions could be considered, such as SMS notifications or in-app notifications with user confirmation.

Redundancy and High Availability:
Decision: To achieve the uptime requirement, the system will use an active redundancy method with distributed services and a load balancer to maintain service operation in case of maintenance or other reasons.
Justification: Redundancy and high availability are crucial for minimizing downtime and ensuring uninterrupted service. By employing distributed services and a load balancer, the system can distribute the workload and redirect traffic in case of failures or maintenance, providing a seamless experience for users.
Alternative Approach: Implementing a cloud-based infrastructure using a platform-as-a-service (PaaS) provider, such as Amazon Web Services (AWS) or Google Cloud Platform (GCP), can offer built-in redundancy and high availability features. These platforms often provide automatic scaling and load balancing capabilities, reducing the need for manual configuration and maintenance.

Concurrent Processing for Transaction Demand:
Decision: The system will make use of concurrent processing wherever possible to meet the required transaction demand.
Justification: Concurrent processing allows multiple transactions to be executed simultaneously, improving system throughput and responsiveness. It ensures that the system can handle a high volume of transactions efficiently.
Alternative Approach: Employing asynchronous processing or message queuing systems, such as Apache Kafka or RabbitMQ, can provide scalability and handle peak loads effectively. These systems decouple the processing of requests from the immediate response, allowing for parallel processing and reducing the impact of resource contention.

5. Architectural Mechanisms

Scripting Platform

This mechanism provides a fast and flexible platform for web scraping. Since publicly available APIs are not assumed to be present, the scripting platform allows the development of custom solutions to scrape data from various price sources. It enables tailored scraping logic for each source, facilitating the monitoring of a wide range of products and sources.

Inversion of Control

Inversion of Control is a pattern provided by the Spring Framework. It promotes loose coupling and modular design by allowing dependencies to be injected into a generic program. IoC enhances the flexibility, extensibility, and testability of the application by decoupling components and facilitating the management of dependencies.

Availability

Availability refers to the system being reachable and in working condition. To ensure availability, the system must remain operational during software updates, component crashes, and other unforeseen events. By employing fault-tolerant mechanisms and redundant components, the system can maintain its availability even in the face of failures or maintenance activities.

Security

Security mechanisms are crucial to protect the system and its data. The storage of security-critical information, such as credit card details and passwords, must be safeguarded against potential attacks. Encryption techniques ensure the confidentiality and integrity of sensitive data, while authentication and authorization mechanisms control access to prevent unauthorized actions. Additionally, all communications between users and the system should be encrypted using HTTPS to secure data transmission over public networks.

Scalability

Scalability is essential to handle the increasing customer base and growing transaction volumes. The architecture should support the ability to scale horizontally by adding more resources or vertically by enhancing existing resources to meet the demands of the expanding business. By ensuring response times remain within acceptable limits and designing the system for easy expansion, scalability can be achieved without compromising performance.

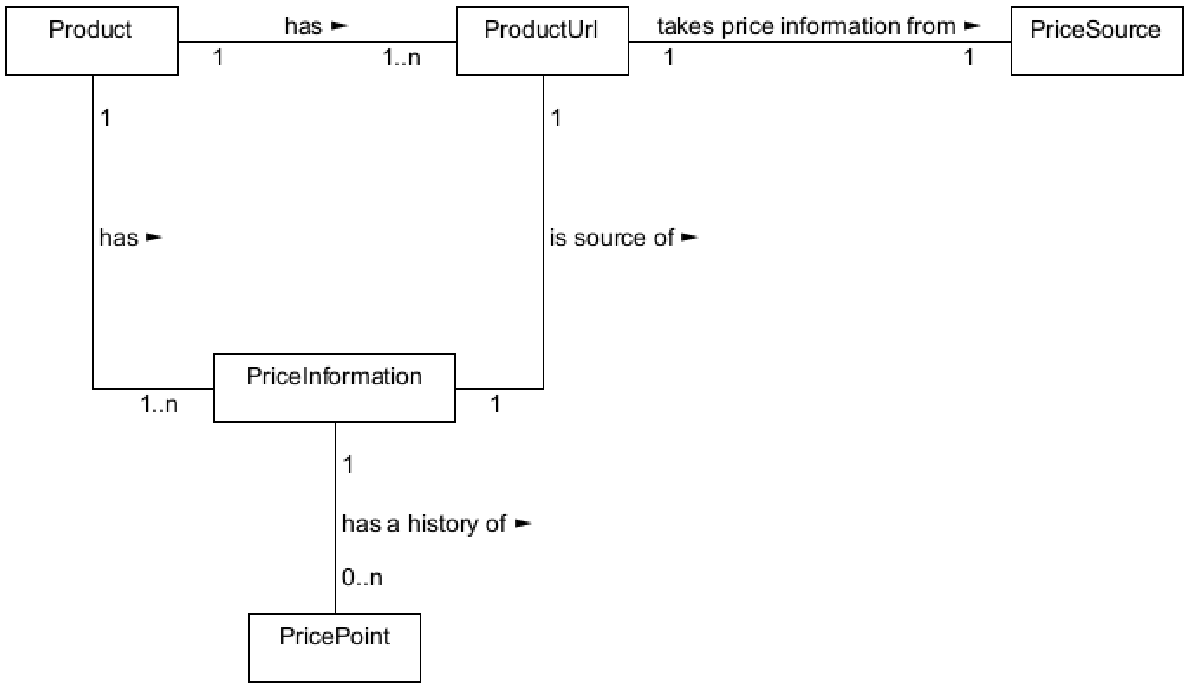
Concurrent Processing

Concurrent processing enables the system to handle multiple transactions simultaneously, improving throughput and responsiveness. By utilizing techniques like asynchronous processing or message queuing systems, the system can efficiently process requests in parallel, reducing bottlenecks and enhancing performance during peak loads.

6. Key abstractions

Users:
Users are individuals who interact with the AnyChange system. There are different types of users, including normal users, seller users, and admin users, each with specific roles and permissions. Users are authenticated to access the system and perform actions based on their authorized privileges.

Products:
Products represent the items available for monitoring within the AnyChange system. Each product has various attributes such as name, description, and category. Products can be associated with multiple price sources, allowing



users to track price variations from different sources.

Price Sources:
Price sources represent external platforms or sources from which price data is collected. Each product can have multiple price sources associated with it. Price sources provide information on price points for products at different times, enabling users to monitor price changes across various sources.

Price Information:
Price information includes the current price and price history of a product obtained from different price sources. It captures the fluctuations and trends in product prices over time, allowing users to make informed decisions based on historical data.

7. Layers or architectural framework

Presentation Layer:

The presentation layer is responsible for the user interface and interaction with the system. It includes components such as web pages, forms, and user input mechanisms. The presentation layer communicates with the underlying layers to retrieve and display relevant data to users.

Application Layer:

The application layer contains the business logic and core functionalities of the AnyChange system. It handles user requests, processes data, and orchestrates the interactions between various components. The application layer implements the Model-View-Controller (MVC) pattern to separate concerns and ensure modular development.

Model Layer:

The model layer represents the data model and the entities within the system. It includes the abstractions discussed earlier, such as products, users, price sources, product URLs, price points, and price information. The model layer defines the structure, behavior, and relationships of these entities, as well as the operations performed on them.

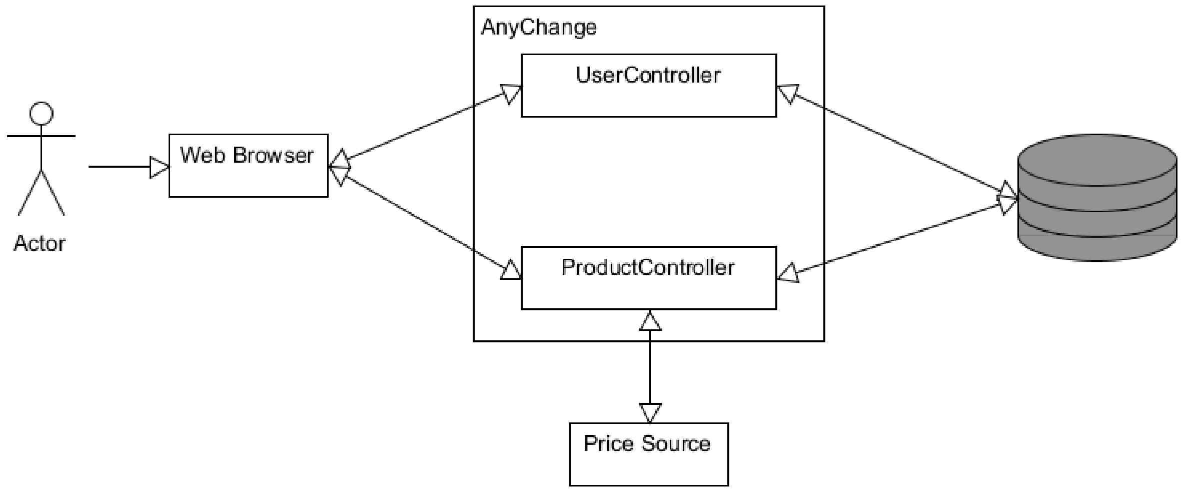
Persistence Layer:

The persistence layer handles the storage and retrieval of data from the underlying database system. It provides mechanisms for interacting with the database, executing queries, and performing CRUD (Create, Read, Update, Delete) operations. The persistence layer ensures data integrity and consistency throughout the system.

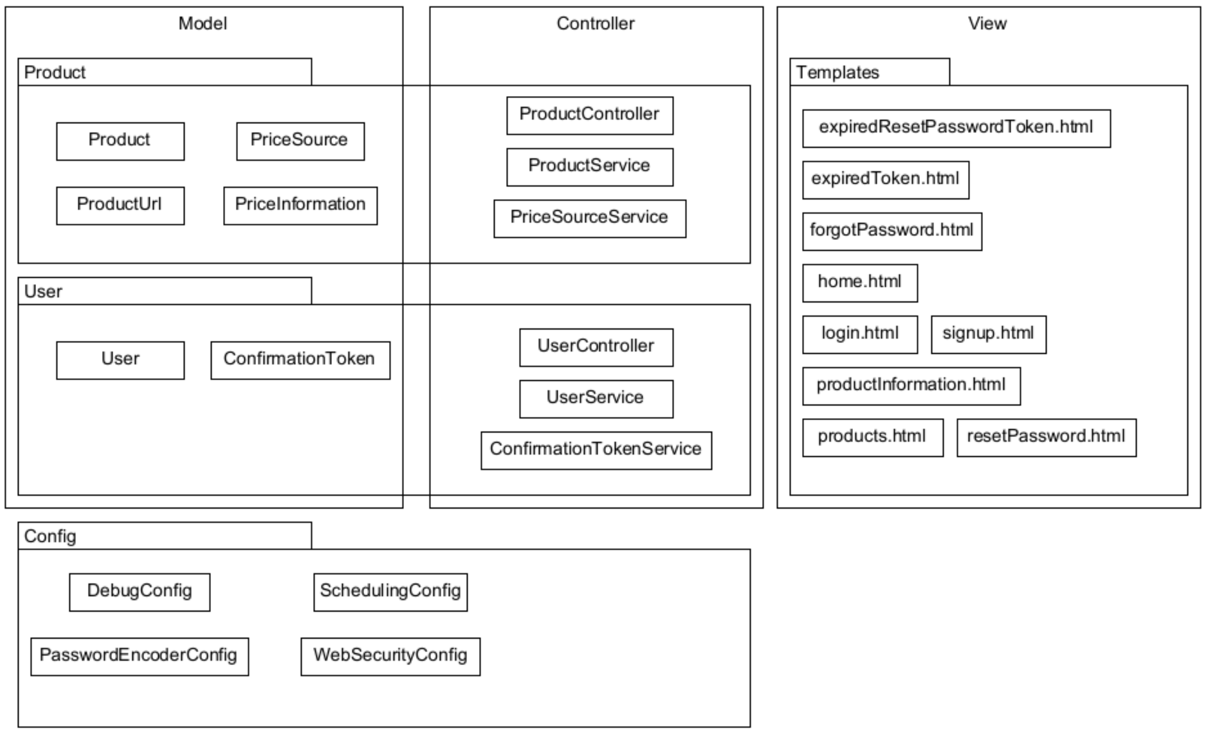
External Services Layer:

The external services layer encompasses any external systems, APIs, or third-party integrations utilized by the AnyChange system. This layer facilitates interactions with external platforms for tasks such as web scraping, data synchronization, or authentication. The external services layer enables seamless integration with external sources of product data and ensures smooth communication with external systems.

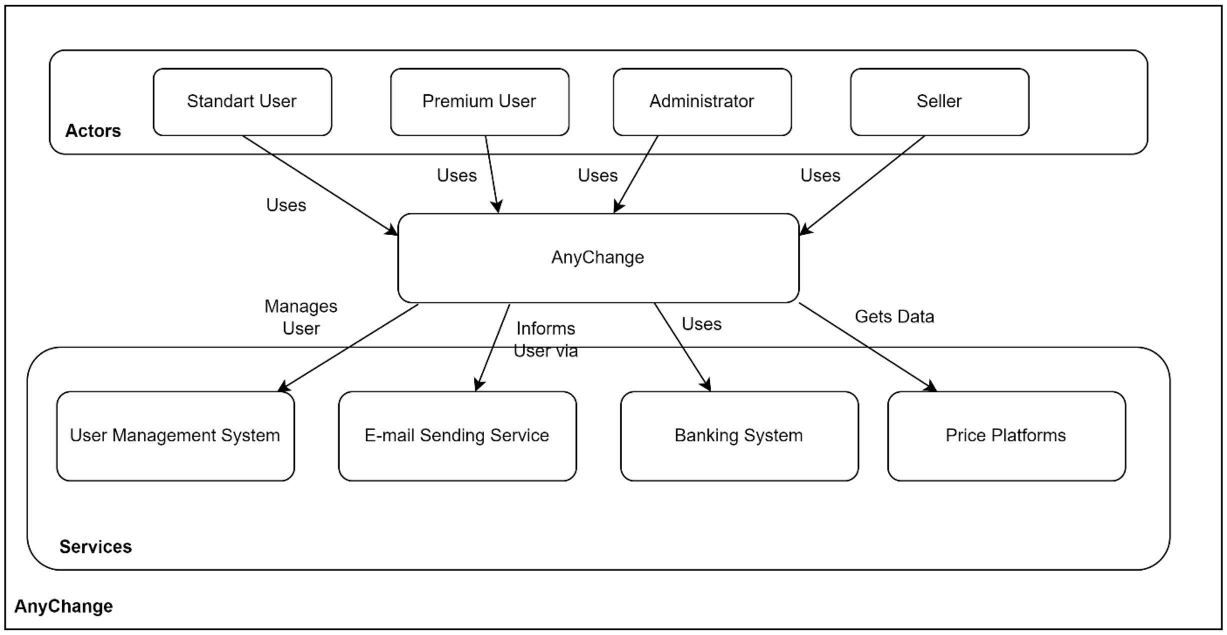
The layered architecture of AnyChange promotes modularity, separation of concerns, and flexibility. Each layer has defined responsibilities and interfaces, allowing for independent development, testing, and maintenance. The layers collaborate to fulfill user requests, process data, and deliver a responsive and robust system.



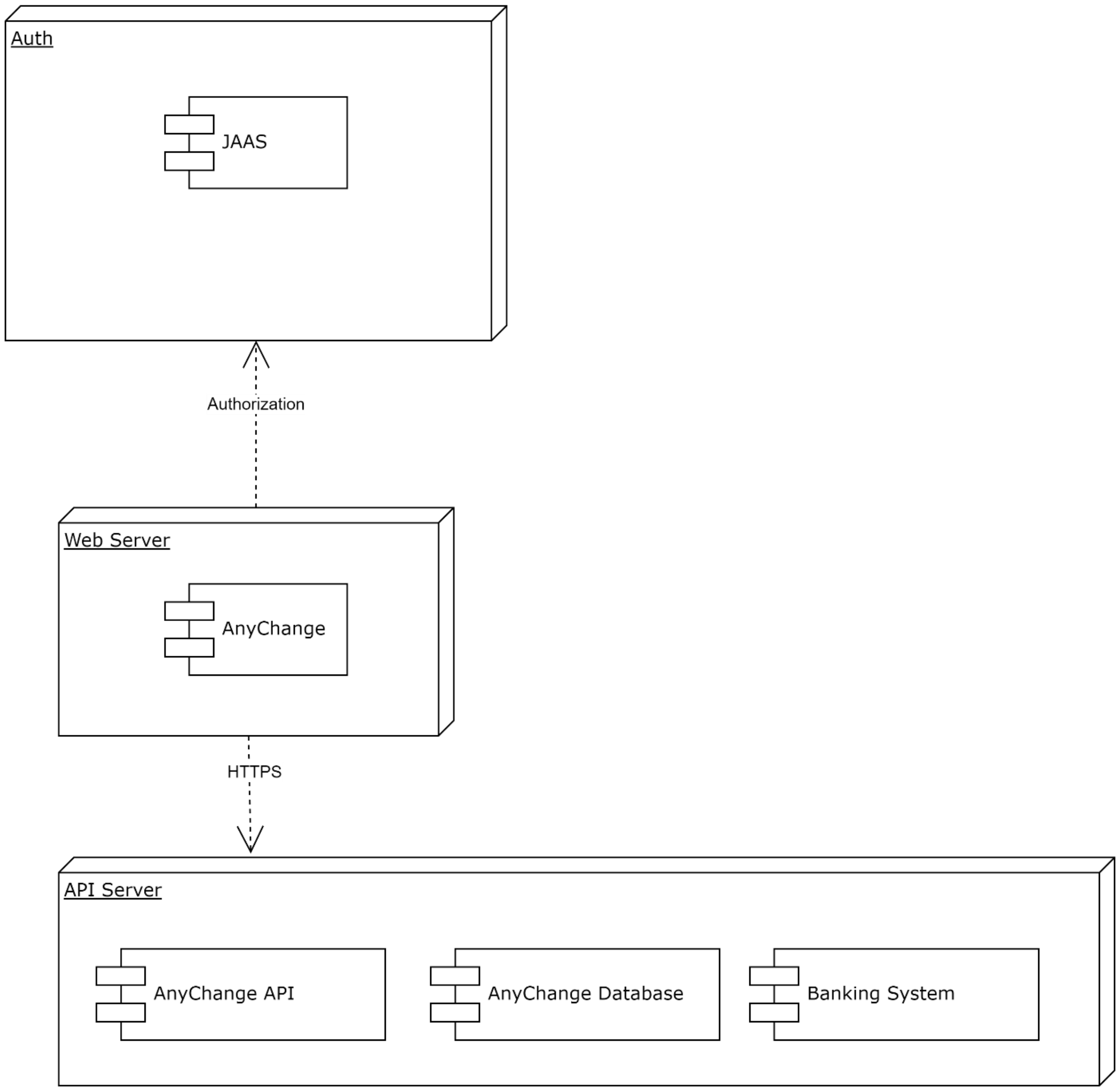
8. Architectural views



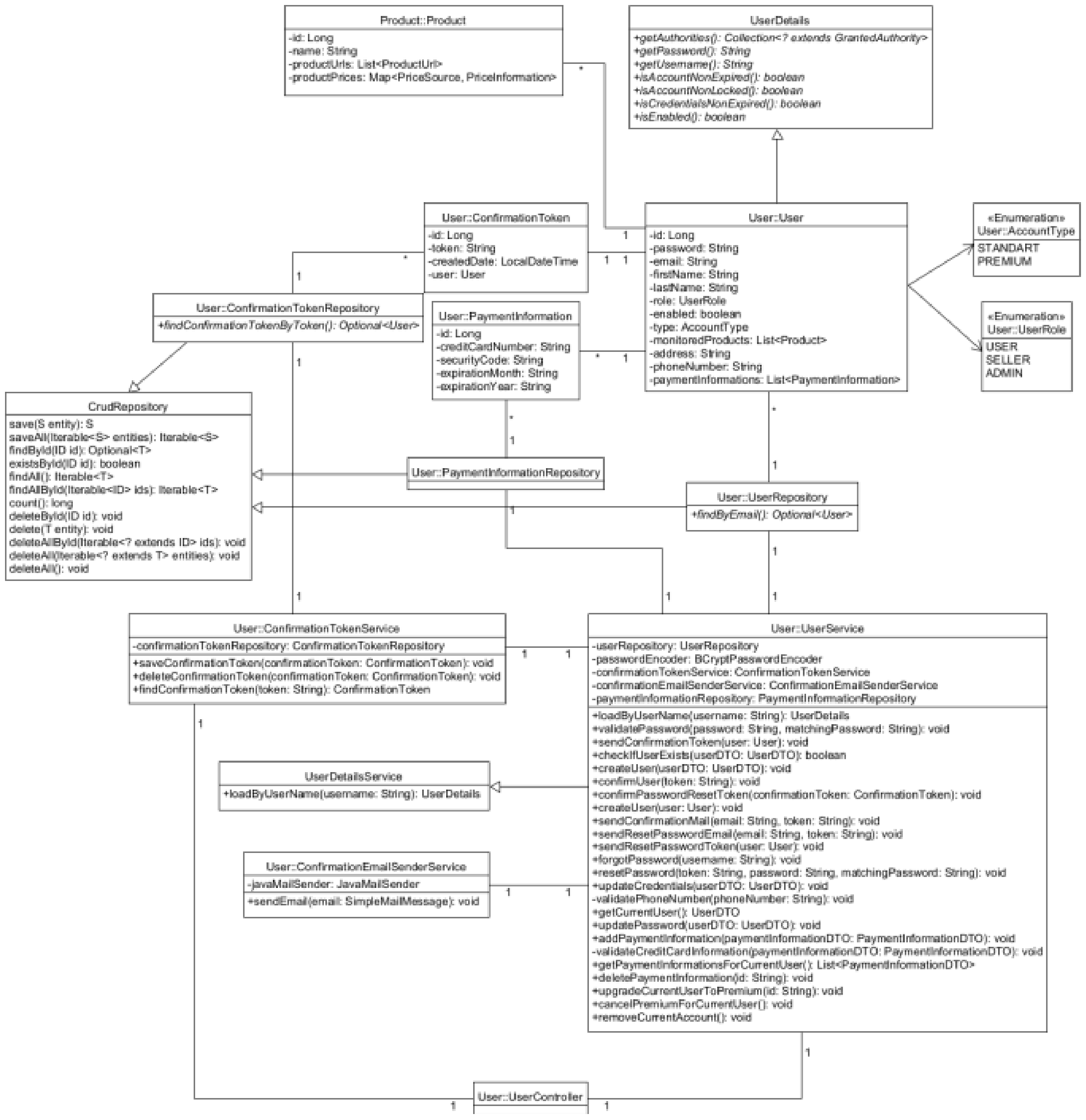
8.1 Context View



8.2 Physical View



8.3 Logical View



Product::Product
-id: Long
-name: String
-productUrls: List<ProductUrl>
-productPrices: Map<PriceSource, PriceInformation>
UserDetails
+getAuthorities(): Collection<? extends GrantedAuthority>
+getPassword(): String
+getUsername(): String
+isAccountNonExpired(): boolean
+isAccountNonLocked(): boolean
+isCredentialsNonExpired(): boolean
+isEnabled(): boolean
User::ConfirmationToken
-id: Long
-token: String
-createdDate: LocalDateTime
-user: User
User::User
-id: Long
-password: String
-email: String
-firstName: String
-lastName: String
-role: UserRole
-enabled: boolean
-type: AccountType
-monitoredProducts: List<Product>
-address: String
-phoneNumber: String
-paymentInformations: List<PaymentInformation>
«Enumeration»
User::AccountType
STANDART
PREMIUM
«Enumeration»
User::UserRole
USER
SELLER
ADMIN
User::ConfirmationTokenRepository
+findConfirmationTokenByToken(): Optional<User>
User::PaymentInformation
-id: Long
-creditCardNumber: String
-securityCode: String
-expirationMonth: String
-expirationYear: String
CrudRepository
save(S entity): S
saveAll(Iterable<S> entities): Iterable<S>
findById(ID id): Optional<T>
existsById(ID id): boolean
findAll(): Iterable<T>
findAllById(Iterable<ID> ids): Iterable<T>
count(): long
deleteById(ID id): void
delete(T entity): void
deleteAllById(Iterable<? extends ID> ids): void
deleteAll(Iterable<? extends T> entities): void
deleteAll(): void
User::PaymentInformationRepository
User::UserRepository
+findByEmail(): Optional<User>
User::ConfirmationTokenService
-confirmationTokenRepository: ConfirmationTokenRepository
+saveConfirmationToken(confirmationToken: ConfirmationToken): void
+deleteConfirmationToken(confirmationToken: ConfirmationToken): void
+findConfirmationToken(token: String): ConfirmationToken
User::UserService
-userRepository: UserRepository
-passwordEncoder: BCryptPasswordEncoder
-confirmationTokenService: ConfirmationTokenService
-confirmationEmailSenderService: ConfirmationEmailSenderService
-paymentInformationRepository: PaymentInformationRepository
+loadByUserName(username: String): UserDetails
+validatePassword(password: String, matchingPassword: String): void
+sendConfirmationToken(user: User): void
+checkIfUserExists(userDTO: UserDTO): boolean
+createUser(userDTO: UserDTO): void
+confirmUser(token: String): void
+confirmPasswordResetToken(confirmationToken: ConfirmationToken): void
+createUser(user: User): void
+sendConfirmationMail(email: String, token: String): void
+sendResetPasswordEmail(email: String, token: String): void
+sendResetPasswordToken(user: User): void
+forgotPassword(username: String): void
+resetPassword(token: String, password: String, matchingPassword: String): void
+updateCredentials(userDTO: UserDTO): void
-validatePhoneNumber(phoneNumber: String): void
+getCurrentUser(): UserDTO
+updatePassword(userDTO: UserDTO): void
+addPaymentInformation(paymentInformationDTO: PaymentInformationDTO): void
-validateCreditCardInformation(paymentInformationDTO: PaymentInformationDTO): void
+getPaymentInformationsForCurrentUser(): List<PaymentInformationDTO>
+deletePaymentInformation(id: String): void
+upgradeCurrentUserToPremium(id: String): void
+cancelPremiumForCurrentUser(): void
+removeCurrentAccount(): void
UserDetailsService
+loadByUserName(username: String): UserDetails
User::ConfirmationEmailSenderService
-javaMailSender: JavaMailSender
+sendEmail(email: SimpleMailMessage): void
User::UserController