Software Architecture Document

Table of Contents

1. Introduction

The Software Architecture Document provides an overview of the architecture for the Lock Settings Configuration. This document is intended to communicate the architecture of the system to stakeholders, including developers, testers, and project managers. It outlines the high-level structure and components of the system, as well as key design decisions and rationale.

2. Requirements Specification:

Design Requirement #	Requirement
FR 1.0	**Platform Compatibility**: The app should be compatible with both iOS and Android mobile devices, ensuring a consistent user experience across different platforms.
FR 2.0	**User Authentication:** The app should provide user authentication functionality to ensure that only authorized service technicians can access the configuration features.
FR 2.1	**Lock Configuration Retrieval:** The app should fetch lock configuration data from the provided mock API (https://run.mocky.io/v3/d5f5d613-474b-49c4-a7b0-7730e8f8f486) upon user request.
FR 2.2	**Display Lock Configuration**: Once the lock configuration data is retrieved, the app should display the configuration parameters for both primary and secondary doors separately.
FR 3.0	**Parameter Adjustment**: The app should allow the service technician to adjust various parameters related to the installation of locks on double doors, such as lock type, door thickness, strike plate size, bolt length, etc.

FR 3.1	**Save Configuration**: The app should allow the user to save the configured parameters for future reference or sharing with other team members.
FR 3.3	**Independent Configuration**: Each parameter adjustment should be independent for the primary and secondary doors, allowing different configurations for each door leaf within the double door unit.
FR 4.0	**Real-time Updates**: The app should reflect any parameter adjustments in real-time, providing immediate feedback to the user.
FR 4.1	**Error Handling**: The app should handle errors gracefully, displaying meaningful error messages to the user in case of failed API requests or other issues.
FR 4.2	**Offline Support**: The app should have offline support, allowing users to access previously saved configurations and make adjustments even when not connected to the internet.
FR 4.3	**Security**: The app should implement appropriate security measures to protect user data and ensure secure communication with the API.

2. Architectural Goals and Constraints

- **Multi-platform Support**: The focus of this architecture to use single shared code base to build application for both Android and iOS platform.
- **Scalability**: The architecture should support scaling to accommodate increased usage and data volume.
- **Reliability**: The system should be reliable, minimizing downtime and data loss.
- **Security**: Security measures should be implemented to protect sensitive data and prevent unauthorized access.
- **Maintainability**: The architecture should facilitate ease of maintenance and future enhancements.
- **Performance**: The system should perform efficiently to meet user expectations.
- **Interoperability**: Integration with external systems should be seamless and well-defined.

- **Technology Stack Constraints**: The system must be developed using Java/.Net (TBD) for the backend and Android Jetpack Compose for the frontend.

3. Architectural Overview

The system follows a microservices architecture, comprised of the following major components:

- **User Interface**: The frontend application developed using Android Jetpack Compose with Kotlin, responsible for presenting the user interface and interacting with users.

- **API Gateway**: Acts as a single entry point for clients to access various microservices.

- **Microservices**:
  - **User Service**: Manages user profiles and authentication tokens.
  - **Product Service**: Handles product catalog and inventory management.
  - **Order Service**: Manages order processing and fulfillment.

- **Database Layer: TBD and Redis for caching.**

4. Architectural Decisions

- **Microservices Architecture**: Chosen to enable scalability, maintainability, and flexibility in development and deployment.

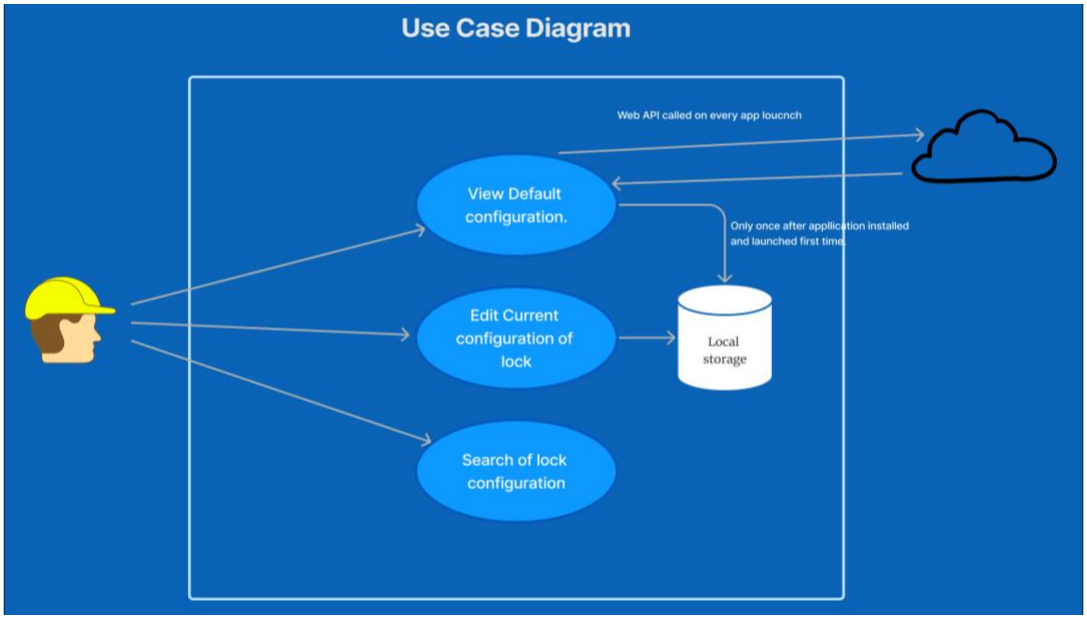
- **API Gateway**: Implemented to provide a unified interface for clients and to handle cross-cutting concerns such as authentication and rate limiting.

- **Containerization with Docker**: Used to package each microservice and its dependencies into a container for easy deployment and scalability.

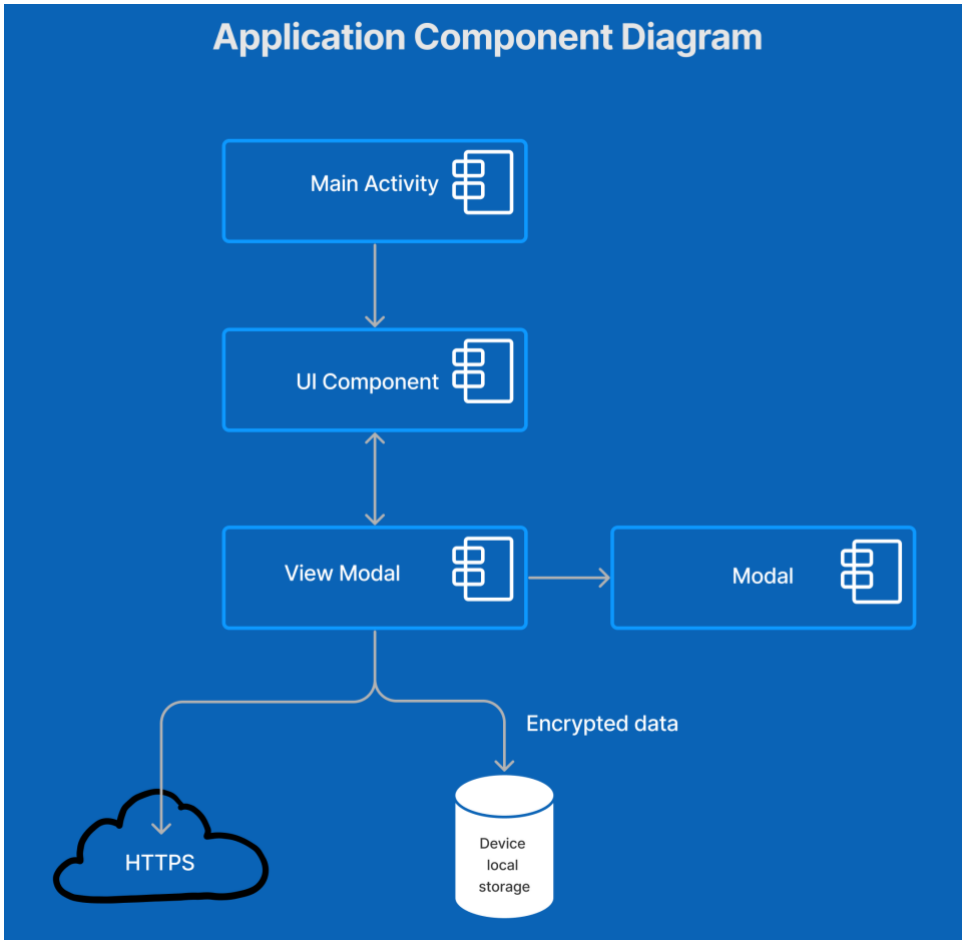
- **Event-Driven Architecture**: Adopted for asynchronous communication between microservices, enhancing scalability and decoupling.

5. Architectural Views and Diagrams

5.1 Use Case Diagram:



5.2 Component Diagram:



MVVM Design Pattern: This helps separate your UI (Views/Composables) from the business logic (Model) and presentation logic (ViewModel). This leads to cleaner, more maintainable code.

Composable Functions: These are the building blocks of your UI in Jetpack Compose. They allow for declarative UI development, making it easier to describe what the UI should look like.

Mutable State: This is a state management solution from Jetpack Compose. It holds the state of your UI properties and triggers a recomposition of the UI whenever the state changes.

Retrofit: This is a popular HTTP client library for Android that simplifies making network calls and parsing JSON responses.

Encrypted Shared Preferences: This is a secure way to store user configuration details on the device. It ensures that sensitive information is not stored in plain text.

5.3 Flow Diagram

The MainActivity class in this project will sets up the UI for your application using Jetpack Compose. It's a bit complex to represent in a class diagram directly. However, I can provide a simplified breakdown of the main components and how they interact:

MainActivity: This is the entry point of your Android application. It sets up the UI using Jetpack Compose.

AccessControlConfigurationViewMobal: This seems to be a ViewModel class that provides access to data related to access control configurations.

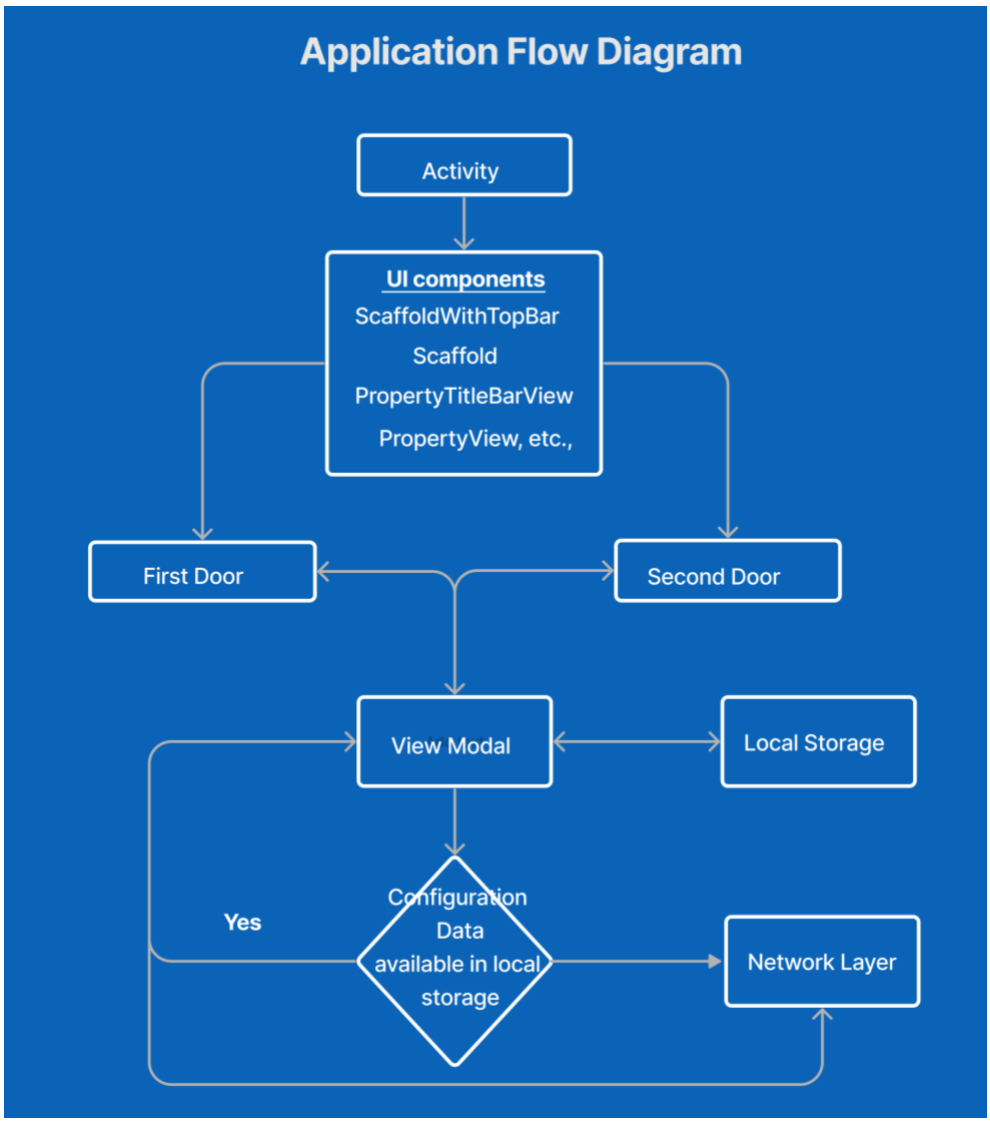
AccessControlConfigurationModal: This represents the data model for access control configurations.

LockConfiguration: This is an interface or superclass implemented by various configuration classes like LockAngle, LockType, LockVoltage, etc.

ScaffoldWithTopBar: This is a Composable function that represents the overall layout structure of your UI, including a top app bar and the main content area.

PropertyView: This is a Composable function that represents a single property view in your UI. It seems to display information about a particular lock configuration property.

PropertyTitleBarView: This is a Composable function that represents a title bar view in your UI, likely showing the title for the properties being displayed.



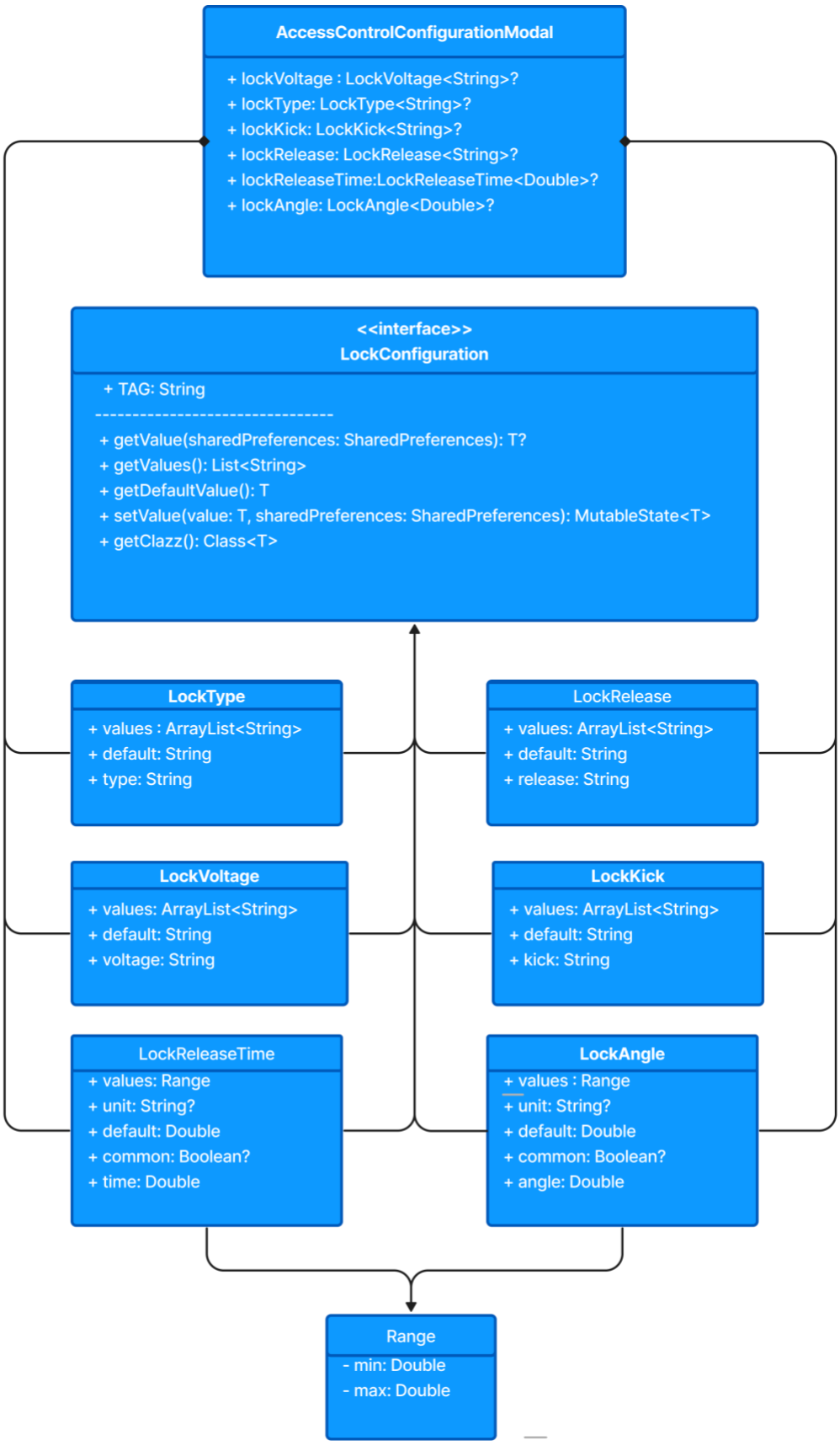
buildSearchBar: This is a Composable function that represents a search bar in your UI for filtering properties.

Various Jetpack Compose UI components like Column, Spacer, LazyColumn, CircularProgressIndicator, etc., are used to structure and display the UI elements.

Given the complexity and dynamic nature of Jetpack Compose UI, representing it in a traditional class diagram might not fully capture its essence. However, you can break down your UI components into smaller functional units and represent their relationships and interactions more clearly.

5.4 Modal Class Diagram:

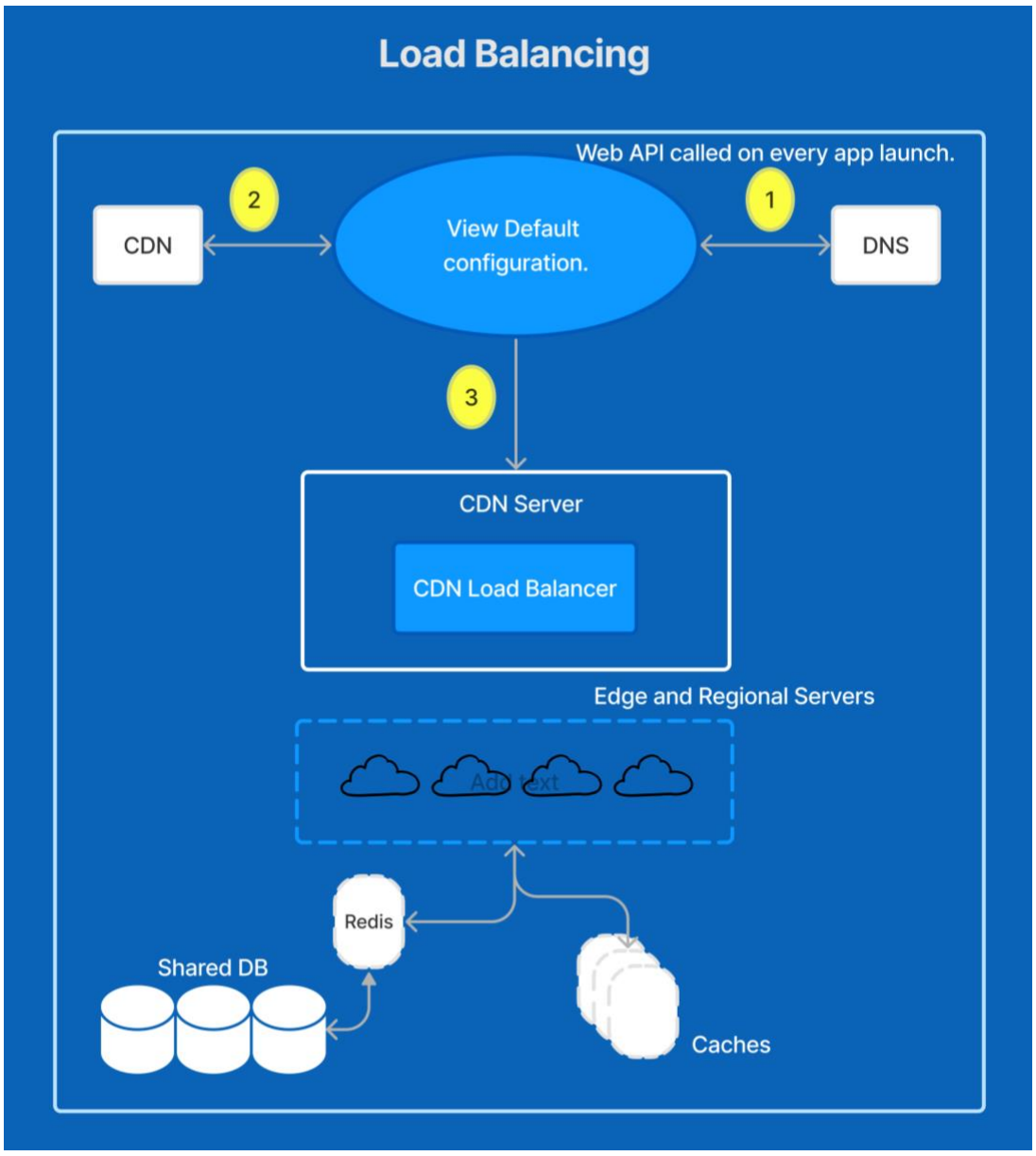
- AccessControlConfigurationModal is the main class.
- It has properties such as lockVoltage, lockType, lockKick, lockRelease, lockReleaseTime, and lockAngle, each of which corresponds to another class (LockVoltage, LockType, LockKick, LockRelease, LockReleaseTime, and LockAngle respectively), with generic types.



- The "?" symbol indicates that these properties can be nullable.
- Properties are prefixed with + to denote that they are public. However, in Kotlin, by default, properties are public.

5.5 Load Balancing

If your application is about to serve a few hundred users with no prospects of significant growth, scalability might be an unreasonable waste of time and money. However, it's a must-have for a modern startup or business that is about to increase the number of customers and serve a large audience's needs.



CDN: A Content Delivery Network (CDN) is a network of distributed servers strategically positioned across various geographic locations. The primary purpose of integrating a CDN into our architecture is to enhance the performance, scalability, and security of our web-based applications and services.

DNS: Serves several key purposes within our architecture, DNS translates domain names (e.g., example.com) into IP addresses (e.g., 192.0.2.1), allowing clients to locate and connect to network resources using meaningful domain names.

Load Distribution: DNS load balancing distributes incoming traffic across multiple servers or endpoints, enhancing fault tolerance, scalability, and performance.
Redundancy and Failover: DNS redundancy and failover mechanisms ensure high availability and fault tolerance by directing traffic to alternate servers or locations in the event of server failures or network outages.

Security: DNS security features, such as DNSSEC (DNS Security Extensions) and DNS filtering, protect against various threats, including DNS spoofing, cache poisoning, and malware attacks.

REDIS: Remote Distributary Service is used to cache frequently accessed data, such as database query results, API responses, and rendered web pages, reducing database load and improving application performance.

6. Quality Attributes

- **Performance**: Response times should be within acceptable limits even under peak loads.
- **Reliability**: Minimize downtime and ensure data consistency.
- **Scalability**: Able to scale horizontally to handle increased load.
- **Security**: Implement robust authentication and authorization mechanisms to protect user data. [If needed]
- **Maintainability**: Code should be well-structured and documented to facilitate ongoing maintenance and updates.

7. Risks and Mitigation Strategies

- **Vendor Lock-in**: Mitigated by adhering to open standards and using widely adopted technologies.
- **Data Security**: Addressed through encryption, access controls, and regular security audits.
- **Scalability Challenges**: Addressed through horizontal scaling and load balancing strategies.

8. Glossary

- **Microservices**: Architectural style that structures an application as a collection of loosely coupled services.
- **API Gateway**: Entry point for clients to access backend services.
- **Event-Driven Architecture**: Architectural pattern where components communicate asynchronously via events
- **Kotlin** : Kotlin is a modern, statically-typed programming language that runs on the Java Virtual Machine (JVM), as well as on other platforms such as Android, JavaScript.
- **Swift** : Programming language developed by Apple for building applications for macOS, iOS, watchOS, anjkd tvOS.
- **Kotlin Multiplatform** : It is a technology introduced by JetBrains that allows developers to write code that can be shared across multiple platforms, such as Android, iOS, web, desktop, and backend.