Software Design Documentation

Genius Smart Lock Data Platform Management System

Renee Singh, Katie Kodama, Carmen Weatherman, Celestine Le

Introduction

A 2021 study found that more than 12 million households had smart locks installed in their homes.[1] Rising crime rates are driving consumer demand for advanced home security measures: the U.S. smart lock market is predicted to reach a market value of ~USD 2.32 Billion by 2030.[2] Genius Smart Lock is a third-party smart lock system that strives to provide consumers with seamless integration into their home security ecosystems. Genius Smart Lock's Data Platform Management System (DPMS) will facilitate communication and message processing across internal/external services within this ecosystem to ensure efficient and performant integration.

System Overview

The DPMS will handle events/message-passing between internal and external dependencies through an event-based message queue and scale for millions of residences sending information at a time. The DPMS will directly interface with other systems such as the primary Alarm System, providing APIs to handle data access and transactions. The DPMS aims to provide a scalable, performant message-processing platform, serving reliable and efficient home security to homeowners.

Design Considerations

Dependencies:

- The Genius Smart Lock User Account Management System will be responsible for initiating requests for database access and modification through the DPMS (such as retrieving information about smart locks, users, authorized guests, entry logs, or posting changes such as new registered locks, users, entry logs, or manual deactivation)
- The Home Security System's Data Platform will grant the DPMS access to user information and property information

[1] https://www.parksassociates.com/blogs/home-systems-and-controls/rise-of-smart-locks-and-smart-access-control8
[2] https://www.grandviewresearch.com/horizon/outlook/smart-lock-market/united-states

- The Home Security System Dashboard should be configured for Genius Smart Lock management on the customer portal, allowing the ability to view and change the status of a smart lock
- The Home Security System's Response System should be configured for mediating communications between the smart lock and response actions, from requesting a response to canceling one
- The Home Security System's Alarms should be configured to the smart lock so that it handles cases of unauthorized access or manual deactivation of an alarm
- Integration with government-provided API for emergency responses in cases of destruction of the smart lock or unauthorized access

General Constraints

- The DPMS will adhere to Genius Smart Lock LLC's existing policies, procedures, and protocols.
- The DPMS shall be compliant with regional standards such as the California Consumer Privacy Act and the California Privacy Rights Act through transparent data usage practices, protection of consumer privacy rights, and securing consumers' sensitive data.
- The DPMS will follow NIST standards for preparing baseline security controls, performing risk assessments, and continuous monitoring to protect consumers' information.
- The DPMS will follow OWASP benchmarks to identify and test for security risks affecting our applications and information systems.
- The DPMS must be designed to scale horizontally and vertically, accommodating up to 5 million users and 20 million locks by 2030 without significant degradation in performance.
- The DPMS will be in charge of storing a predicted 25-30 TB worth of data during its first year of operation, and will need the ability to scale its memory capacity to meet fluctuations in demand.
- The DPMS must offer backwards compatibility with legacy software operating within home security systems up to 5 years old.
- The DPMS user interfaces, including mobile and web applications, must comply with WCAG 2.1 Level AA standards to ensure accessibility for users with disabilities.

Goals and Guidelines

- The DPMS must provide an API for configuring a lock's address, health, current status, and passcodes for guests
- The DPMS will provide the ability for authorized users to cancel a response action that was triggered by a false alarm
- The DPMS shall use encryption at rest and in transit to protect all sensitive data
- The DPMS shall have 99.999% availability with about 5.25 minutes of downtime per year

- The DPMS will need to support different versions of firmware and offer backward compatibility with the main Alarm System
- The DPMS will also need to provide backwards compatibility for third party smart home applications and mobile devices that remain in usage among Genius smart lock users
- The DPMS must be capable of verifying user authorization requests and notifying the Home Security System's response team in the case of unauthorized access
- The DPMS must be resilient against DDoS attacks and other influxes of traffic
- The DPMS will be able to handle message failures without disrupting the flow of incoming messages or losing message data
- The DPMS will be able to continue serving users in the event of instance failures without issues such as duplication
- The DPMS must include quarterly updates for security and compatibility improvements and a minimum of 10 years of post-deployment support.
- The DPMS must provide logging and audit trails for all operations, accessible to authorized personnel for at least five years, and include real-time monitoring of bottlenecks.
- The DPMS must maintain an average response time of under 200 milliseconds for all user queries and support at least 10,000 simultaneous operations
- The DPMS must implement a disaster recovery plan with geographically distributed backups and failover mechanisms to restore full functionality within five minutes of catastrophic system failure.

Development Methods

The DPMS will undergo several iterations of testing, starting with automated unit testing with an emphasis on evaluating our system's ability to handle events and process messages properly. Our next priority will be integration testing to ensure separate components of our data platform communicate and interact with each other as intended and that data is being preserved. Once integration testing has confirmed our system is functioning at a high level, we will conduct security testing in accordance with OWASP and NIST benchmarks.

After passing unit, integration, and security testing, the DPMS will be ready for accessibility testing, usability testing, and regression testing. Regression testing will be used to identify any breaking changes that accompany code updates. Any updates within the DPMS will use semantic versioning to communicate changes, their severity, and backwards compatibility status.

For more higher-level alterations, like changes in the API contract, data schema, or addition/deletion of major functions, we will follow a protocol of deprecation rather than removal, announcing these actions via deprecation notices that include migration guidelines (While we had considered the advantage of removals, such as saving storage space and reducing burden to our data platform, we found that this would introduce more breaking changes). Selection of features to be deprecated will be based on developments within the domain of home security and mobile devices, periodic research into which integrations are still in use among our

user base, and, most importantly, any security issues identified as a result of outdated/inefficient code.

Architectural Strategies

The DPMS focuses on scalability, availability, and security so that it can handle large intakes of data (25-30TB) from residences across the U.S., perform as expected when expected, and protect homeowners' sensitive information. The DPMS follows asynchronous event-driven architecture to better handle large volumes of data in real-time. Event-driven architecture also helps decouple components, allowing producers and consumers to be handled independently. This makes future maintenance more efficient and reduces downtime.

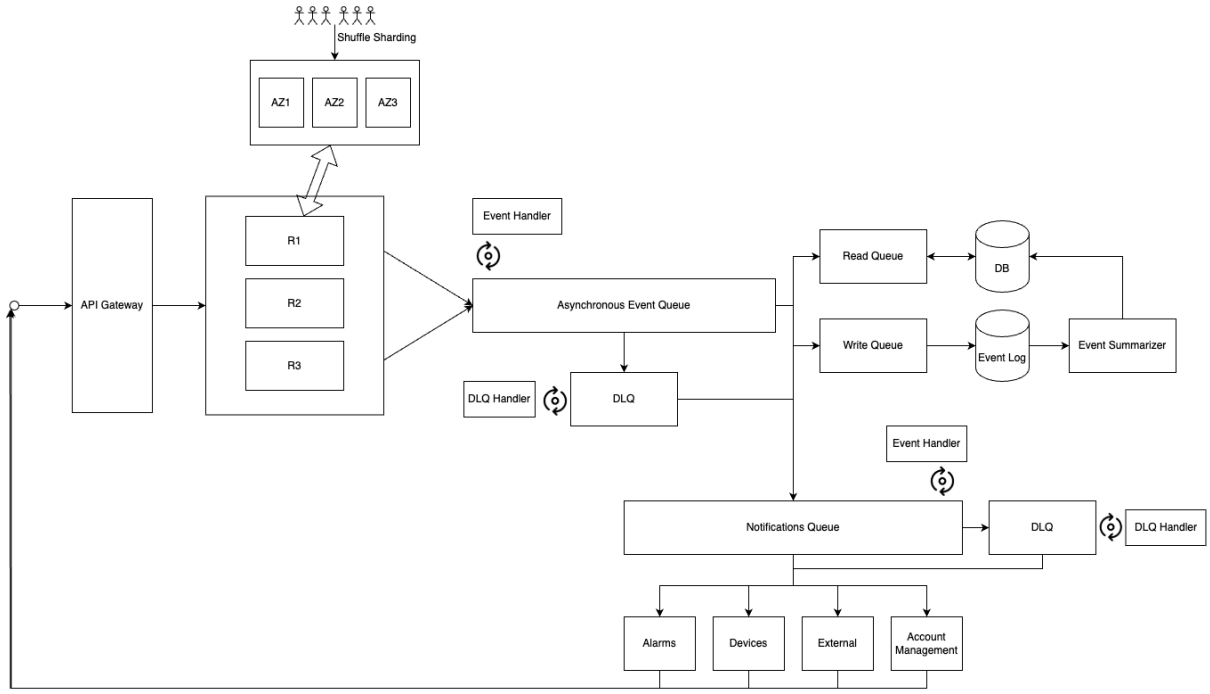
The DPMS also implements redundancy at geographic, software, and data levels by running multiple instances per availability zone per region and storing copies of our data stores to enhance availability and security. In the case of unexpected failures or unauthorized attacks on an instance, zone, or region, the DPMS uses load balancing to redirect traffic to working instances without worrying about data loss while supporting quicker recovery times.

Considering the need for scalability and performance as data is written or read in the DPMS, we employ a CQRS framework so that our data stores for reading vs. writing data are held separately. As the DPMS will be reading data more often than it writes data, this will improve performance and also allow us to optimize database models for reading and writing purposes, such as using a relational database model for easier reading and a non-relational database model for a larger collection of logs.

We utilize a combination of edge caching and database caching to meet our KPI of an average response time under 200 milliseconds. Implementing edge caching can significantly reduce latency for users by storing static assets like firmware updates or user interface files closer to end users in geographically distributed locations. This ensures faster data delivery, particularly for users in remote regions, while also reducing the load on central servers.

For database caching, we plan to use a NoSQL database that can store frequently queried data such as lock statuses, user permissions, or audit logs. This minimizes the frequency and cost of expensive database read operations, allowing the platform to serve real-time requests more efficiently during peak traffic.

System Architecture



The DPMS makes use of asynchronous message queues, event handling, and load-balancing to support scalability and availability of Genius Smart Locks all across the United States. The DPMS also handles security, protecting data and message handling from unauthorized actors.

API Gateway

When a request is made to the DPMS's API, they are received by an API gateway. The API gateway acts as an elastic load balancer, performing throttling in the event of a traffic or scaling-related failure. This throttling will place a restriction on the number of requests accepted for processing so that the failure does not continue to place a greater burden on the system. It will do this in a way that ensures that the rates of incoming read requests and write requests are similar which enables for a smoother recovery in cases of traffic-related failures. The API gateway also authenticates and authorizes requests, routing them to the appropriate instances of the DPMS. Through implementing a gateway, our architecture optimizes scalability so that we can reach our KPI of 10,000 operations at a time.

Multi-Region/Multi-Zone Redundancy

The DPMS will utilize horizontal scaling and cell-based architecture, distributing the system into independent subunits that serve a subset of the user base. Firstly, region isolation will be used by deploying independent instances of the data platform for smart lock users in different geographical areas. Each region will also consist of several availability zones (US West-1, US West-2), and users within regions will be further segmented using shuffle sharding– randomly assigned to overlapping resource subsets. Similarly, in the case of unanticipated outages (potentially resulting from natural disasters, fires, or malicious actors), we will further utilize redundancy and horizontal scaling to mitigate outages affecting only one region/data center, redirecting traffic to another functioning region. Multi-region cloud support is crucial to mitigate the impact of physical or hardware issues, reduce downtime, and introduce more reliability and availability. Especially in the case of a natural disaster impacting one region, multi-region support will minimize the loss of important data such as entry logs, while supporting recovery. These measures work together to help the DPMS meet our KPI of 99.999% availability year-round.

Message Brokering

Requests are then passed to a message broking system, starting with being added to an asynchronous event queue, facilitated by an event handler that follows a publisher-subscriber model. The event handler IDs and labels each request based on its purpose, and then tracks its position in the queue to ensure delivery to the next module. In the case of a dead request, the event handler routes it to a dead-letter queue equipped with its own event handler to ensure delivery. The asynchronous event queue routes requests to three other queues: the read queue, write, queue, and notify queue. Based on the request made, the message broker choreographs events in a specific order. For example, in the case of a sensor sending a request to the data platform about unauthorized access, a notification will immediately be sent to relevant sub-systems before waiting for the write system to log it in the database, while in other cases like authorized entry or exit, eventual consistency is acceptable. The notify queue also has its own event handler and DLQ, performing the same responsibilities and routing these requests to the event queues of various different services in the system, such as the Genius Smart Lock User Account Management, Devices and Sensors and Alarms systems, and connected external services such as a Home Security system.

CQRS Database Operations

To decouple database operations, the DPMS will utilize a CQRS model that separates events into a read and write queue. The asynchronous event queue publishes events to the write queue, adding to an append-only event log. The event log is then passed to an event summarizing module, that transforms the data into read-only materialized views– a relational database for user

and smart lock information, and a non-relational format for access logs. These views are horizontally sharded based on the zip codes of each lock, being allocated to the relevant DPMS instance. Separating read and write requests further balances overall load, optimizing the speed of database operations.

Tradeoffs & Bottlenecks

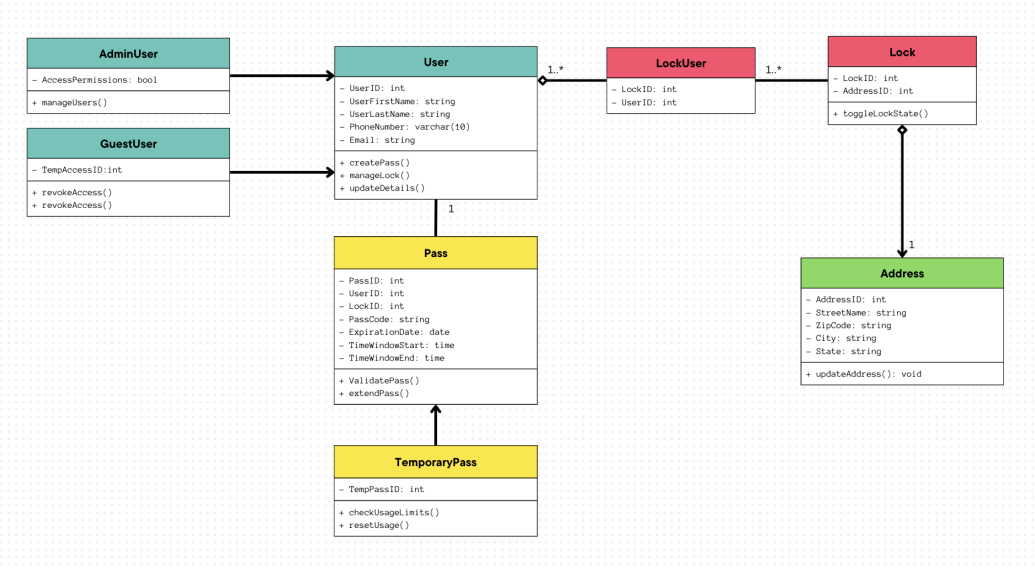
We considered using CRUD operations, where both read and write events would be carried out on the same data model. The benefit of this method is that it would be more resource-efficient, and simpler to implement and maintain due to its single data model. However, a CQRS approach better suits the DPMS's scaling needs, optimizing complex read and write operations as separate concerns, enhancing security of transactions and boosting independent scalability for each concern. However, in choosing CQRS over CRUD, we had to consider added complexity to our design and eventual consistency as primary tradeoffs.

Additionally, we considered routing processed events through a separate event-driven architecture as an alternative to routing them through our existing event handlers. The former enhances decoupling of those processes but introduces added latency and requires more time to implement. We instead prioritized performance over decoupling by eliminating the need for a separate architecture and helping the DPMS meet our KPI of <200 millisecond response time, while also cutting down time needed for implementation.

Our architecture relies on a pub-sub model with an asynchronous event queue, which itself is a bottleneck in our design. However, it proves advantageous because it contributes to increased cohesion within our architecture while decoupling our specific event queues.

Detailed System Design

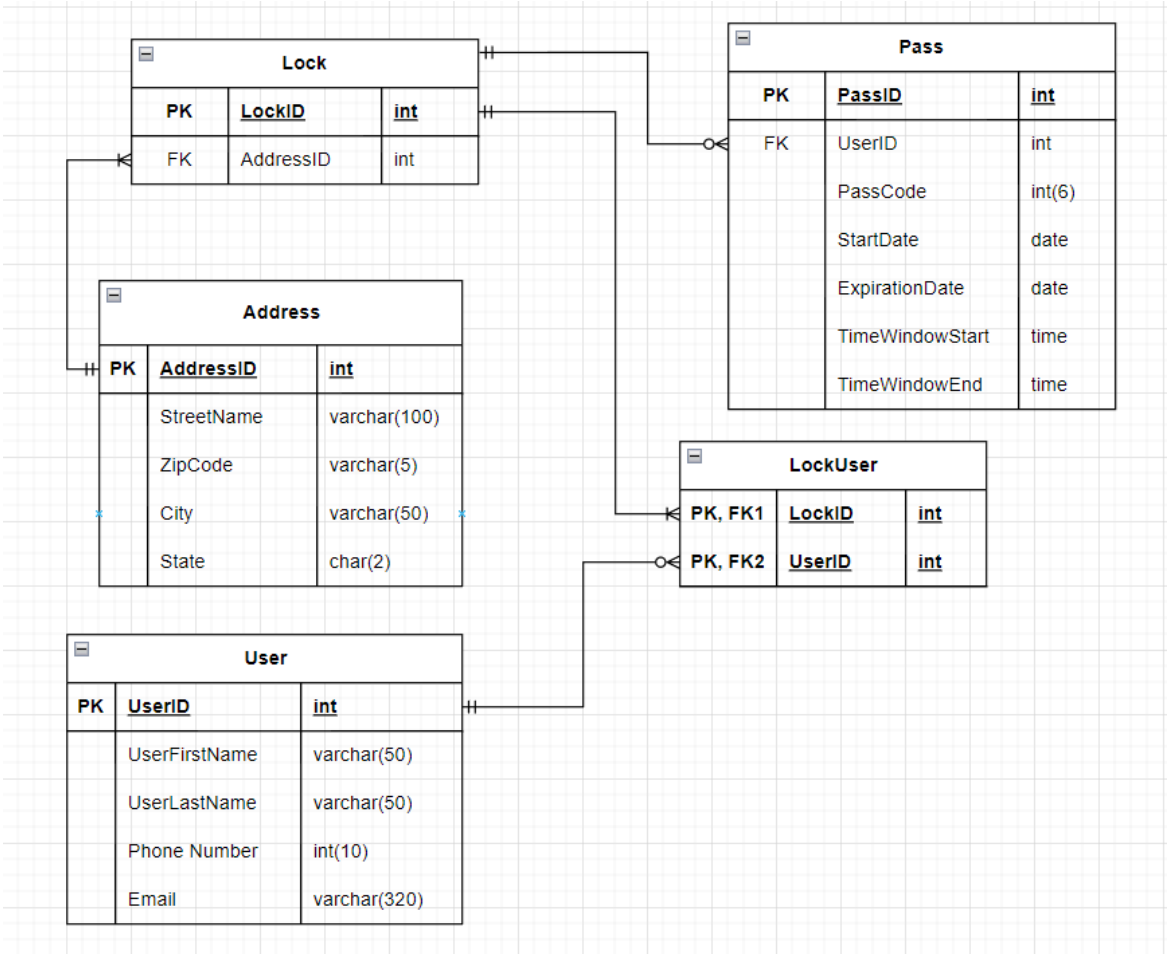
Class Diagram



Our DPMS design organizes users, locks, and passes into entities with clear relationships. The User class connects to multiple locks through LockUser, which tracks user-lock associations. Pass objects allow users temporary access, while Address stores lock locations. Users can be AdminUser or GuestUser, with admins managing locks and users, while guests have limited, temporary access. This structure ensures scalability, secure access management, and flexible interactions between entities.

Methods in the system manage user actions and lock states. The User class has methods like createPass() to generate passes, manageLock() to control lock states, and updateDetails() for personal info changes. The Pass class includes validatePass() and extendPass() for passcode validation and extension. The Lock class has toggleLockState() for lock state changes, and the Address class offers updateAddress() to modify location details. These methods ensure secure, efficient system operations and user interactions.

Database Schema - Relational



This database schema models the structure of the data stored in the system, organizing it into tables that capture the key entities and their relationships. The **User Table** stores personal details like the user's ID, name, phone number, and email, with the UserID as the primary key. The **Lock Table** represents the smart locks, linking each lock to a specific address through the AddressID foreign key, with LockID as the primary key. The **Pass Table** contains information about guest passcodes, including the user and lock they are associated with, along with passcode details, validity period, and time constraints, using PassID as the primary key and foreign keys linking to UserID and LockID. The **Address Table** holds the address details for each lock, with AddressID as the primary key. Finally, the **LockUser Table** normalizes the many-to-many relationship between users and locks through an associative entity, allowing each user to be linked to multiple locks and vice versa, with a composite primary key consisting of both LockID and UserID.

Database Schema - Non-Relational

Our non-relational database schema organizes data using time logs to structure entries and events. The document includes the following entities: ID, timestamp, user, and lock, which distinguish the events being recorded. Each entry has a unique ID, along with a timestamp, user, and log, to document events in case the data is needed. Within the User entity, the associated properties are the ID and full name. The Lock entity contains the properties ID and address.

```
{
    "title": "Record of employee",
    "description": "This document records logs of locks successfully
                    opened",
    "type": "object",
    "properties": {
        "id": {
            "description": "A unique identifier for a log",
            "type": "number"
        },
        "timestamp": {
            "description": "Date and time of the entry log",
            "type": "datetime"
        },
        "user": {
            "description": "user associated with entry log",
            "type": "object",
            "properties" : {
                "id": {
                    "description": "A unique identifier for the user",
```

```
                    "type": "number"
                },
                "fullname": {
                    "description": "full name of the user",
                    "type": "string"
                }
            }
        },
        "lock": {
            "description": "lock that was accessed",
            "type": "object",
            "properties" : {
                "id": {
                    "description": "A unique identifier for the lock",
                    "type": "number"
                },
                "address": {
                    "description": "house address the lock is located at"
                    "type": "string",
                }
            }
        }

    }
}
```

API Contracts

The DPMS will interact with the main Home Security System by integrating with external sources through a multitude of defined API contracts. These contracts specify the interactions between the two systems, noting the expected inputs and outputs for each API call. For example, our DPMS may receive lock status updates, user management data, or potential alarm system alerts via APIs from the home security system.

For instance, the DPMS supports functionality such as creating user passes, managing locks, and processing alarm alerts through methods like createPass() or manageLock(). Each of these methods maps directly to API endpoints. For example, createPass() is associated with a POST /users/:userid/locks/:lockid/passes endpoint, enabling the creation of access passes tied to specific locks. When the endpoint receives pass details such as passcode, startDate, and

expirationDate, the DPMS processes this information within the Genius Smart Lock platform, granting or modifying lock access accordingly.

- User Management:
    - Users can be registered via the POST /users endpoint, which requires details such as name, email, and phone number. Successful creation returns a confirmation message, while errors, such as invalid data, result in 400 responses.
    - Retrieving user information is facilitated through the GET /users/:id endpoint, with error handling for non-existent users.
    - The PATCH /users/:id endpoint allows for updates to user information, ensuring user data remains current.
    - Users can be deleted from the system using DELETE /users/:id, with successful removal indicated by a 204 status code.
- Lock & Pass Management:
    - Lock management includes retrieving all locks associated with a user (GET /users/:id/locks), registering a new lock (POST /users/:id/locks), and deleting a lock from a user's account (DELETE /users/:userid/locks/:lockid).
    - Passes, which grant or restrict access to locks, can be retrieved via the GET /users/:userid/locks/:lockid/passes/:passid endpoint, created with POST /users/:userid/locks/:lockid/passes, or updated/deactivated through PATCH /users/:userid/locks/:lockid/passes/:passid. These endpoints ensure that pass management is efficient and secure.
- System-Wide Operations:
    - All users in the system can be listed using GET /users, while all locks can be retrieved with GET /locks.
    - Specific lock and pass details are accessible via GET /locks/:id and GET /passes/:id, respectively, ensuring visibility across the platform.
    - Entry and exit logs, vital for tracking system activity, can be retrieved using GET /logs and filtered by user ID, lock ID, or address ID.
- Log Management:
    - A centralized log structure records interactions, with each log entry containing a unique ID, timestamp, user ID, and lock ID. Logs are critical for auditing system activity, maintaining accountability, and ensuring traceability.

Glossary

API - is a set of protocols, routines, and tools that allow different software applications to communicate with each other by defining how requests and responses should be structured
Availability zone - a separate group of data centers within a region

Database Schema - a structured framework that defines the organization, relationships, and constraints of data within a database, specifying how the data is stored, accessed, and managed.
Dead letter queue - where messages that failed to be processed after N number of attempts are sent
Event Queue - a data structure that stores and manages a sequence of events or tasks in the order they occur
Gateway - an API that mediates communication and redirects traffic between the client and multiple services
Message broker - a component that facilitates communication between different services by routing, transforming, and managing messages between systems
Region - a defined geographic or logical boundary where data, services, or resources are managed and stored

Appendix

Data Platform API Contract

Account Management

Register a new user

POST /users
URL Params: None
Data Params:
{
UserFirstName: string,
UserLastName: string,
Email: string,
PhoneNumber: integer
}
Success Response:

- Code: 201
- Content: { “User Created”}

Error Response (Empty fields):

- Code: 204
- Content: { error: “No Content” }

Error Response (Invalid Data):

- Code: 400
- Content: { error: “Bad Request” }

Retrieve user information

GET /users/:id
URL Params: id=[integer]
Data Params: None
Success Response:

- Code: 200
- Content: { <user_object> }

Error Response:

- Code: 404
- Content: { error : “User does not exist” }

Update user information

PATCH /users/:id

URL Params: id=[integer]
Data Params:
{
UserFirstName: string,
UserLastName: string,
Email: string,
PhoneNumber: integer
}
Success Response:

- Code: 200
- Content: { “User information updated”}

Error Response:

- Code 404
- Content: { error : “User doesn’t exist” }

Delete account

DELETE /users/:id
URL Params: id=[integer]
Data Params: None
Success Response:

- Code: 204
- Content: {“account successfully deleted”}

Error Response:

- Code: 404
- Content: { error: “User doesn’t exist” }

Lock & Pass Management

lock_object =
{
id: integer
street_name: string
zip_code: varchar(10)
city: string
state: string

}

Pass_object =
{
id: integer
passcode: integer(6),
startDate: date,

```
        expirationDate: date,
        timeWindowStart: time,
        timeWindowEnd: time
}
```

Retrieve information on all locks associated with a user

GET /users/:id/locks
URL Params: id=[integer]
Data Params: None
Success Response:

- Code: 200
- Content: {locks: [{<lock_object>}, {<lock_object>}, {<lock_object>}]}

Error Response:

- Code: 404
- Content: { error : “error retrieving locks” }

Register a new lock

POST /users/:id/locks
URL Params: id=[integer]
Data Params:

```
{
        lockid: integer
}
```

Success Response:

- Code: 201
- Content: { <lock_object> }

Delete a lock from a user account

DELETE /users/:userid/locks/:lockid
URL Params: userid=[integer], lockid=[integer]
Data Params: None
Success Response:

- Code: 204
- Content: { <lock_object> }

Error Response:

- Code: 404
- Content: { error : “lock does not exist” }

Retrieve pass information

GET /users/:userid/locks/:lockid/passes/:passid

URL Params: userid=[integer], lockid=[integer], passid=[integer]
Data Params: None
Success Response:

- Code: 200
    - Content: { <pass_object> }

Error Response:

- Code: 404
- Content: { error : “pass not found” }

Create a new pass

POST /users/:userid/locks/:lockid/passes
URL Params: userid=[integer], lockid=[integer]
Data Params:

```
{
        passcode: integer(6),
        startDate: date,
        expirationDate: date,
        timeWindowStart: time,
        timeWindowEnd: time
}
```

Success Response:

- Code: 201
- Content: { <pass_object> }

Change pass information or deactivate pass

PATCH /users/:userid/locks/:lockid/passes/:passid
URL Params: userid=[integer], lockid=[integer], passid=[integer]
Data Params:

```
{
        passcode: integer(6),
        startDate: date,
        expirationDate: date,
        timeWindowStart: time,
        timeWindowEnd: time
}
```

Success Response:

- Code: 200
    - Content: {<pass_object>}

Error Response:

- Code: 404
- Content: { error : “pass does not exist” }

Smart Lock Account Management API Contract

/user

- **User object**:

```
{
  "userID": "string",
  "username": "string",
  "password": "string",
  "email": "string",
  "created_at": "datetime",
  "updated_at": "datetime"
}
```

  - **POST**: /register
    - Request: username, password, email
    - Response: 201 Created OR 400 Bad Request
  - **DELETE**: /{userID}
    - Request:
    - Response: 204 No Content OR 404 Not Found
  - **GET**: /login
    - Request: None (Handled on server)
    - Response: 200 OK OR 404 Not Found
  - **GET**: /logout
    - Request: None (Handled on server)
    - Response: 200 OK OR 404 Not Found
  - **GET**: /{userID}
    - Request: None (Uses Query Parameters)
    - Response: JSON{User object}; code 200 OK OR 404 Not Found
  - **PUT**: /changePassword
    - Request: oldPassword, newPassword
    - Response: 200 OK OR 404 Not Found

/thirdparty

- This will communicate with the Lock Database Team and Security Account Management Team APIs
- **ThirdPartyObject:**

```
{
  "integrationID": "string",
  "serviceName": "string",
  "apiKey": "string",
  "deviceID": "string",
}
```

  - **POST**: /connectService
    - Request: serviceName, apiKey, deviceID

        - Response: 200 OK OR 400 Bad Request
    - **DELETE**: /disconnectService/{serviceName}/{deviceID}
        - Request: None (Uses Query Parameters)
        - Response: 204 No Content OR 404 Not Found
    - **GET**: /manageIntegrationSettings/{deviceID}
        - Request: None (Uses Query Parameters)
        - Response: JSON{integrationSettings}; 200 OK OR 404 Not Found

/payments

- **PaymentMethod object:**

```
{
 "ticketID": "string",
 "user": {
  "userID": "string",
  "username": "string",
  "password": "string",
  "email": "string"
 },
 "description": "string",
 "date": "datetime",
 "status": "string"
}
```

    - **POST**: /addPaymentMethod
        - Request: cardNumber, expiryDate, billingAddress
        - Response: 201 Created OR 400 Bad Request
    - **DELETE**: /deletePaymentMethod/{paymentMethodID}
        - Request: None (Uses Query Parameters)
        - Response: 204 No Content OR 404 Not Found
    - **PUT**: /updatePaymentMethod
        - Request: paymentMethodID, cardNumber, expiryDate, billingAddress
        - Response: 200 OK OR 404 Not Found
    - **PUT**: /manageSubscription
        - Request: subscriptionID, action (upgrade/cancel)
        - Response: 200 OK OR 404 Not Found
    - **GET**: /viewBillHistory/{subscriptionID}
        - Request: None (Uses Query Parameters)
        - Response: JSON{billHistory}; 200 OK OR 404 Not Found

/ticket

- **Ticket object:**

```
{
 "ticketID": "string",
 "user": {
  "userID": "string",
  "username": "string",
  "password": "string",
```

```
    "email": "string"
   },
   "description": "string",
   "date": "datetime",
   "status": "string"
  }
```


- **POST**: /createTicket
  - Request: user, date, description
  - Response: 201 Created OR 400 Bad Request
- **DELETE**: /closeTicket/{ticketID}
  - Request: None (Uses Query Parameters)
  - Response: 204 No Content OR 404 Not Found
- **PUT**: /updateTicket
  - Request: ticketID, status, description
  - Response: 200 OK OR 404 Not Found

/device

- **Device object:**


```
  {
   "deviceID": "string",
   "deviceName": "string",
   "associatedAddress": "string",
   "owner": {
    "userID": "string",
    "username": "string",
    "password": "string",
    "email": "string"
   },
   "accessLogs": [
    {
     "logID": "string",
     "timestamp": "datetime",
     "action": "string"
    }
   ]
  }
```


- **POST**: /addDevice
  - Request: deviceName, associatedAddress, userID
  - Response: 201 Created OR 400 Bad Request
- **DELETE**: /removeDevice/{deviceID}
  - Request: None (Uses Query Parameters)
  - Response: 204 No Content OR 404 Not Found
- **PUT**: /updateDevicesAddress/{deviceID}
  - Request: associatedAddress (Uses Query Parameter)
  - Response: 200 OK OR 404 Not Found
- **POST**: /recordActivity

        - Request: action, timestamp
        - Response: 201 Created OR 400 Bad Request
    - **GET**: /getAccessLogs/{deviceID}
        - Request: None (Uses Query Parameters)
        - Response: JSON{accessLogs}; 200 OK OR 404 Not Found

/accessControl

- **Access Control Object**

```
{
  "accessControlID": "integer",
  "deviceID": "string",
  "userID": "string",
  "accessLevel": "string",
}
```

    - **PUT**: /grantAccess
        - Request: deviceID, userID, accessLevel
        - Response: 200 OK OR 404 Not Found
    - **PUT**: /revokeAccess
        - Request: accessControlID
        - Response: 200 OK OR 404 Not Found
    - **PUT**: /updateAccessLevels
        - Request: accessControlID, accessLevel
        - Response: 200 OK OR 404 Not Found

/session

- **Session Object**:

```
{
  "sessionID": "string",
  "userID": "string",
  "deviceInfo": "string",
  "loginTime": "datetime",
  "logoutTime": "datetime",
  "isValid": "boolean"
}
```

    - **POST**: /startSession
        - Request: userID, deviceInfo
        - Response: 200 OK OR 400 Bad Request
    - **DELETE**: /endSession/{sessionID}
        - Request: None (Uses Query Parameters)
        - Response: 200 OK OR 404 Not Found
    - **GET**: /validateSession/{sessionID}
        - Request: None (Uses Query Parameters)
        - Response: 200 OK OR 404 Not Found