Vendomatic Project

Design

Version 1.1

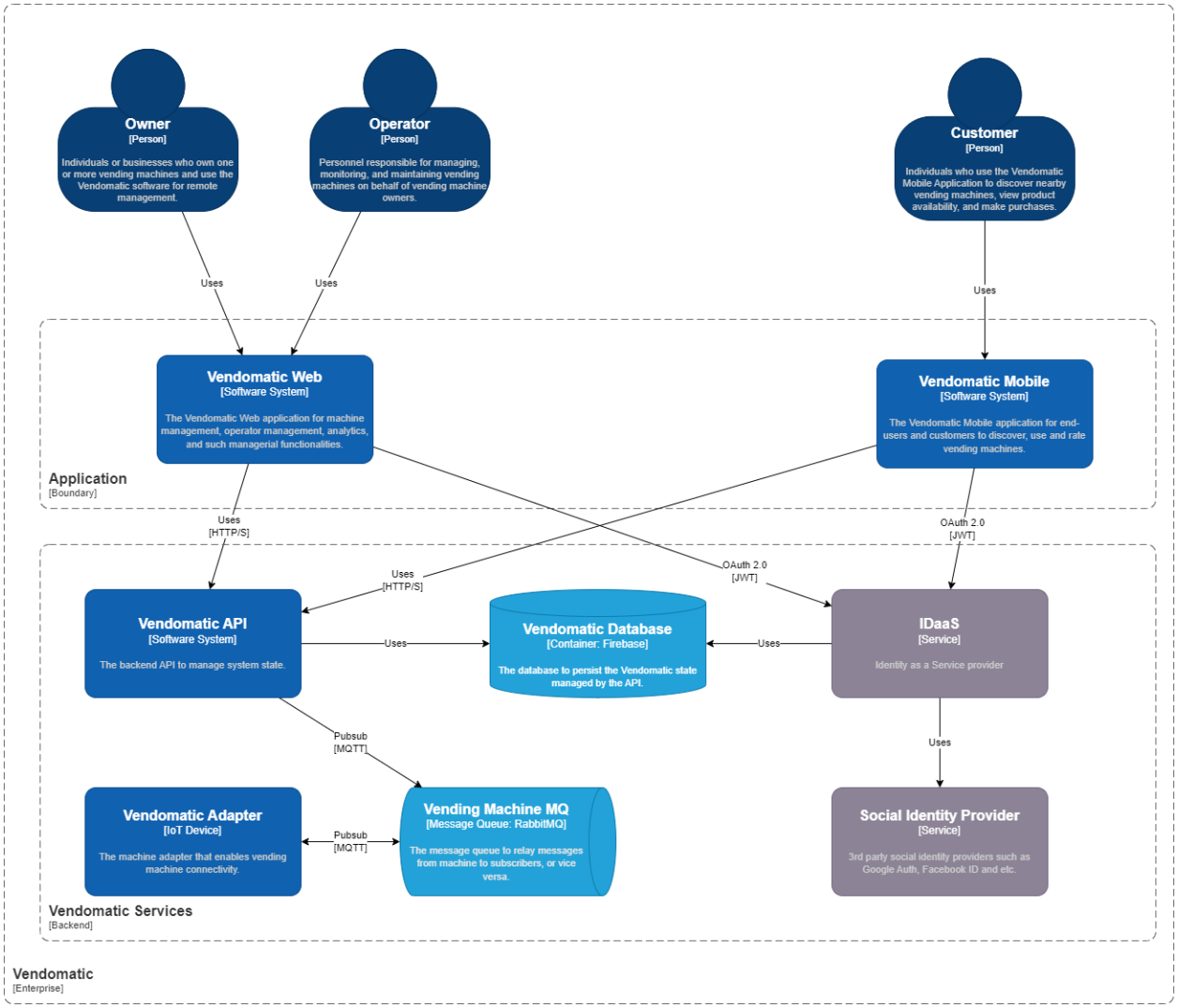
Prepared By: Code Busters

Revision History

Version	Description	Author	Date
0.1	● Draft	Kadir Kılıçoğlu	2023-04-14
0.2	● Review and format	Ezgi Özkan Tuğçe Sözer	2023-04-24
0.3	● Add CQRS pattern ● Add UC2-5 sequence diagrams	Kadir Kılıçoğlu	2023-05-11
1.0	● Realize neighbor machines UC ● Realize rating UC	Kadir Kılıçoğlu	2023-05-30
1.1	● Review and format	Ezgi Özkan Tuğçe Sözer	2023-06-04

Table of Contents

1. Design Structure



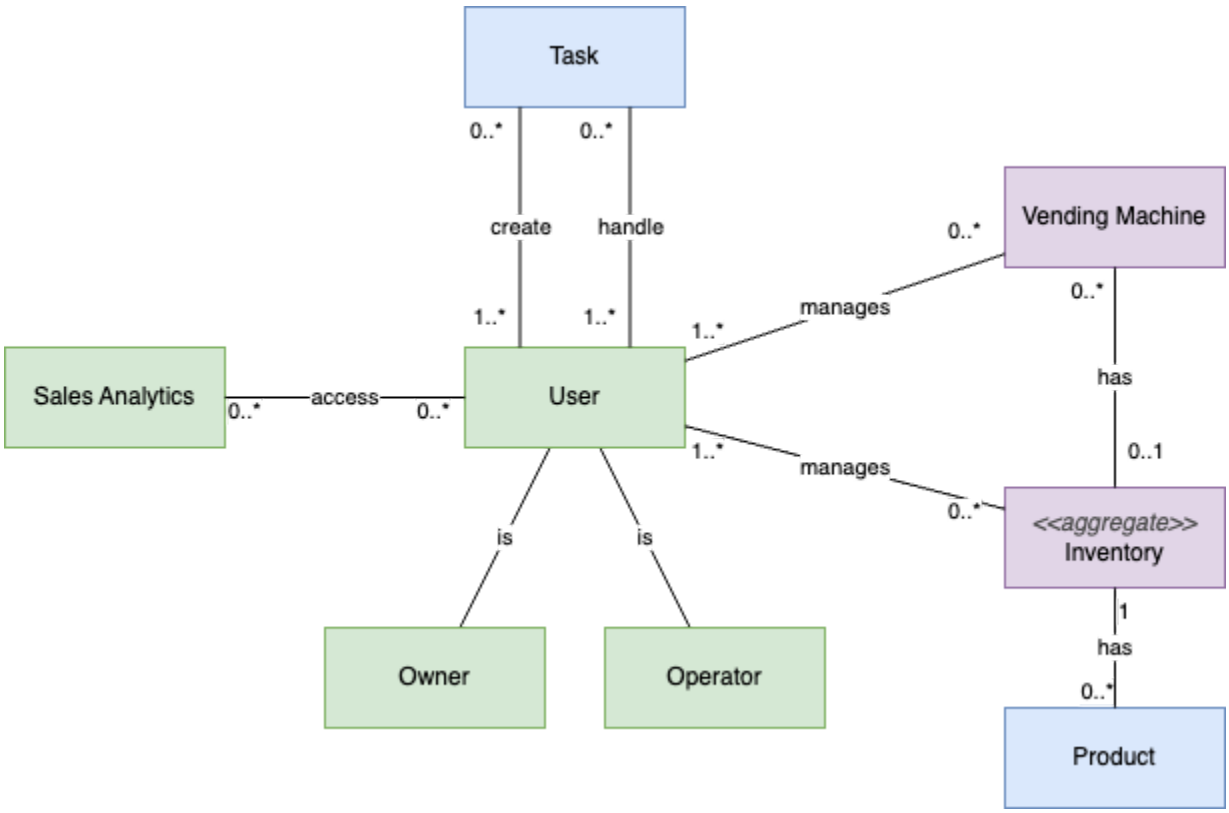
In the Vendomatic enterprise boundary there are basically 3 types of users: the owner, operator, and customer. Owners and operators interact with the web application only within the application boundary whereas the customers only interact with the mobile application.

A 3rd-party IDaaS handles the authentication and authorization, and the same service also handles social identity provider integrations.

Since the basic idea of the Vendomatic software ecosystem is based on the fact that most vending machines are disconnected, the Vendomatic Adapter proxies machine level communication across online channels through MQTT message queues.

2. Subsystems

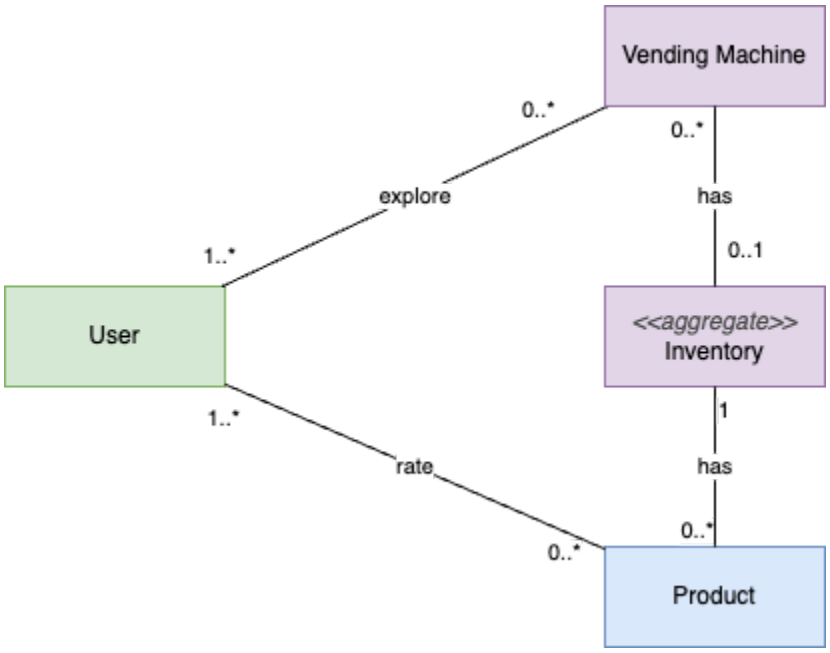
2.1. Vendomatic Web



The Vendomatic web application is a sub-system that covers the management web application for owners and operators. In this direction; a user might be an owner or an operator only. Since this is a management application, the web API supports management of tasks, machines and products over the inventory aggregate.

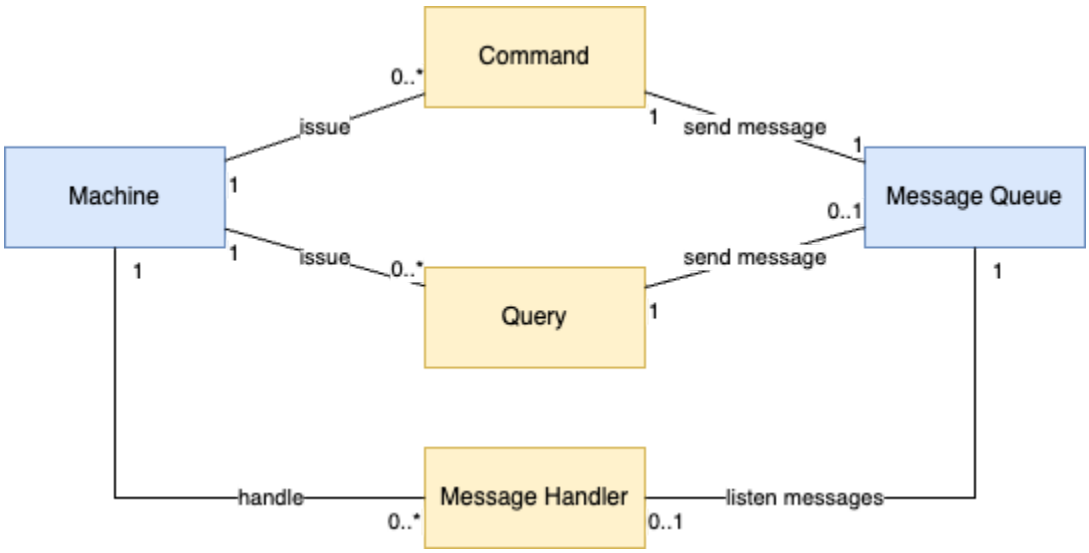
The component model depicts the major components of the Vendomatic web application which complies with the key abstractions described in the Architecture Notebook.

2.2. Vendomatic Mobile

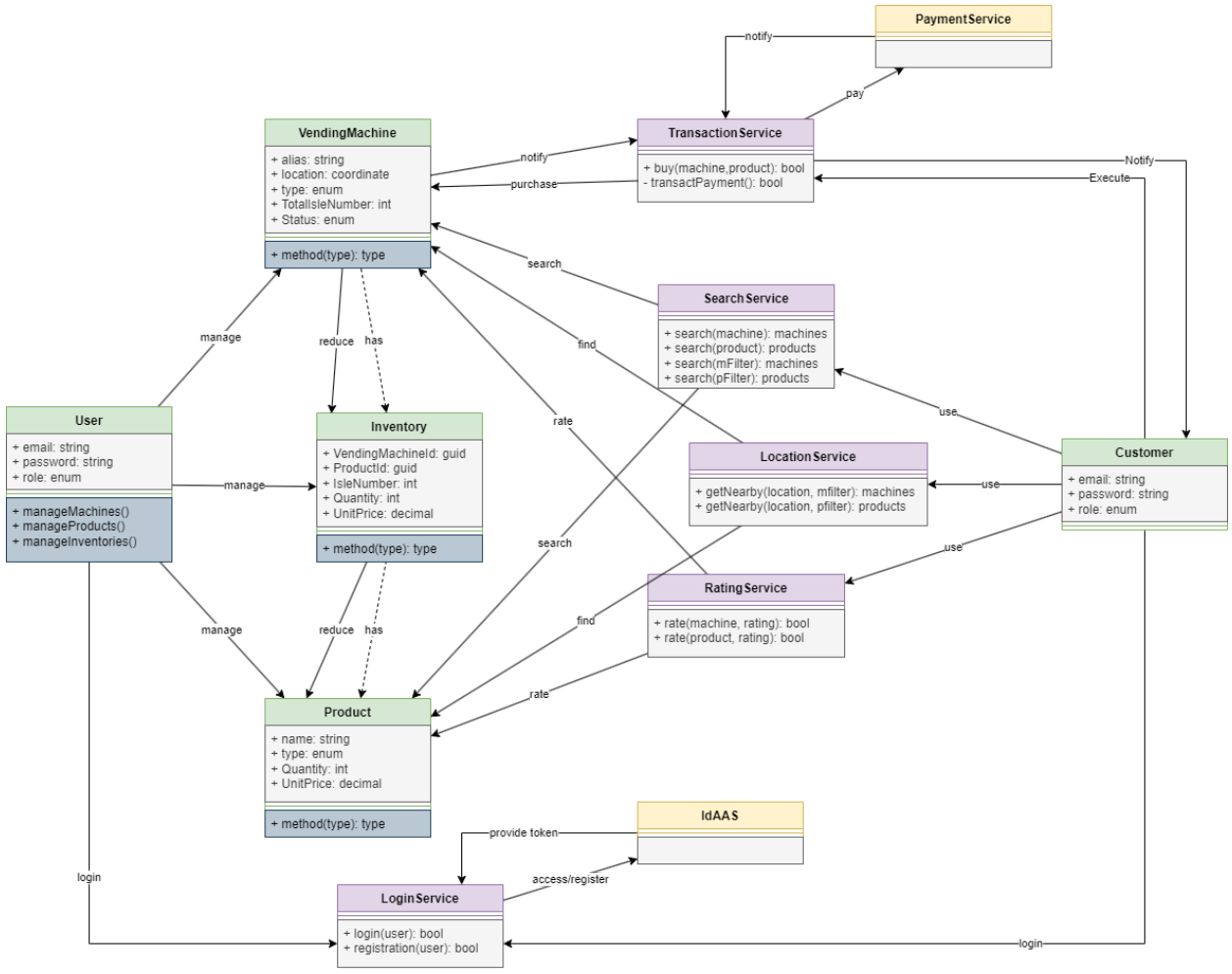


The Vendomatic mobile application targets only customers, hence the components are simplistic by its nature as seen in the component diagram above. Other than the users, the major components are the machines to be discovered and products to be rated or purchased.

2.3. Vendomatic Adapter

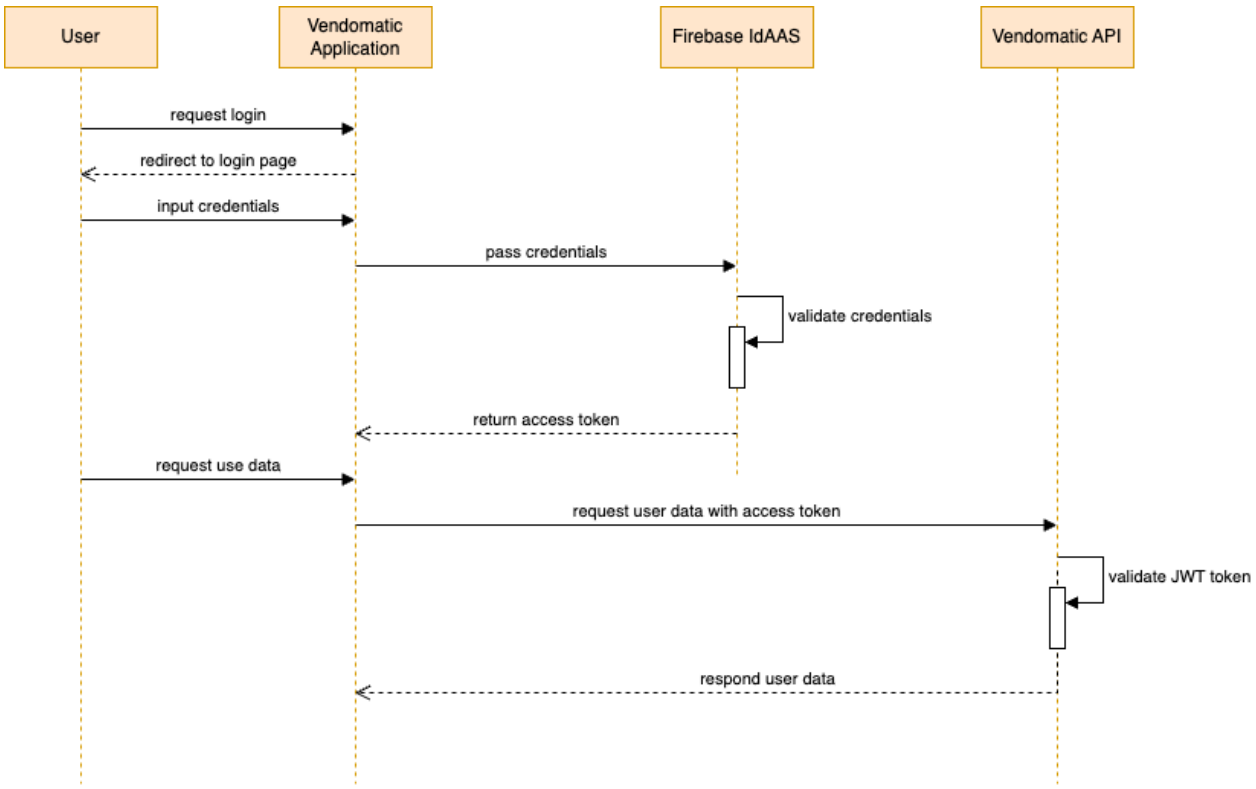


2.4. Vendomatic API



3. Patterns

3.1. Delegated Authentication



The Login flow is a Resource Owner Password Flow pattern common in all sub-systems. In the Resource Owner Password Flow, the client application sends a request to the authorization server that includes the resource owner's username and password. The authorization server verifies the credentials and if they are valid, issues an access token to the client application.

The access token is then used by the client application to access protected resources on behalf of the resource owner. The token is typically sent in the Authorization header of HTTP requests to the resource server.

One advantage of the Resource Owner Password Flow is that it allows clients to obtain access tokens without requiring a separate authentication step.

Overall, the Resource Owner Password Flow provides a simple and straightforward way for trusted client applications to obtain access tokens for accessing protected resources on behalf of the resource owner.

3.2. Command Query Responsibility Segregation - CQRS

Command Query Responsibility Segregation (CQRS) is an architectural pattern that separates read and update operations for a data store. This means that every method should either be a Command which performs an action, or a Query which returns data to the caller, but not both. Commands mutate state and cannot return data, while queries can return data but cannot

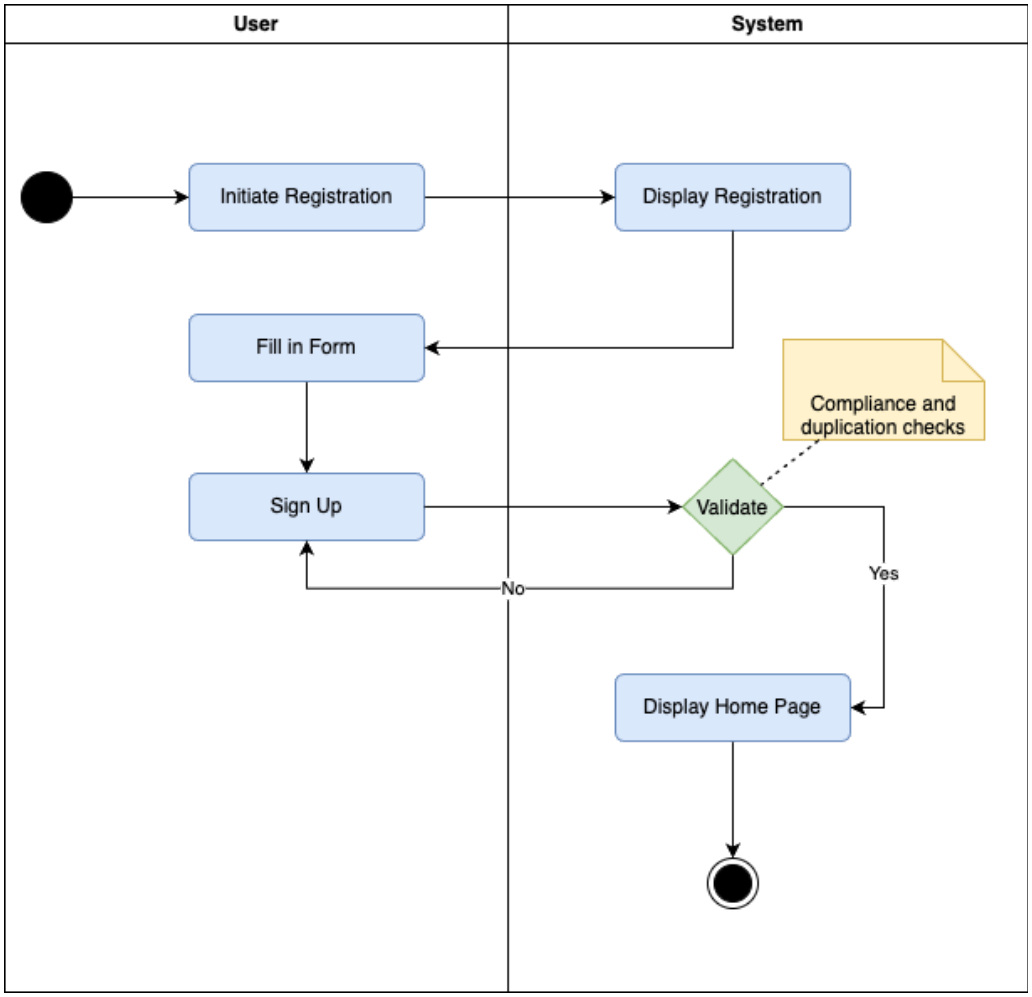
mutate state.

Advantages of CQRS include:

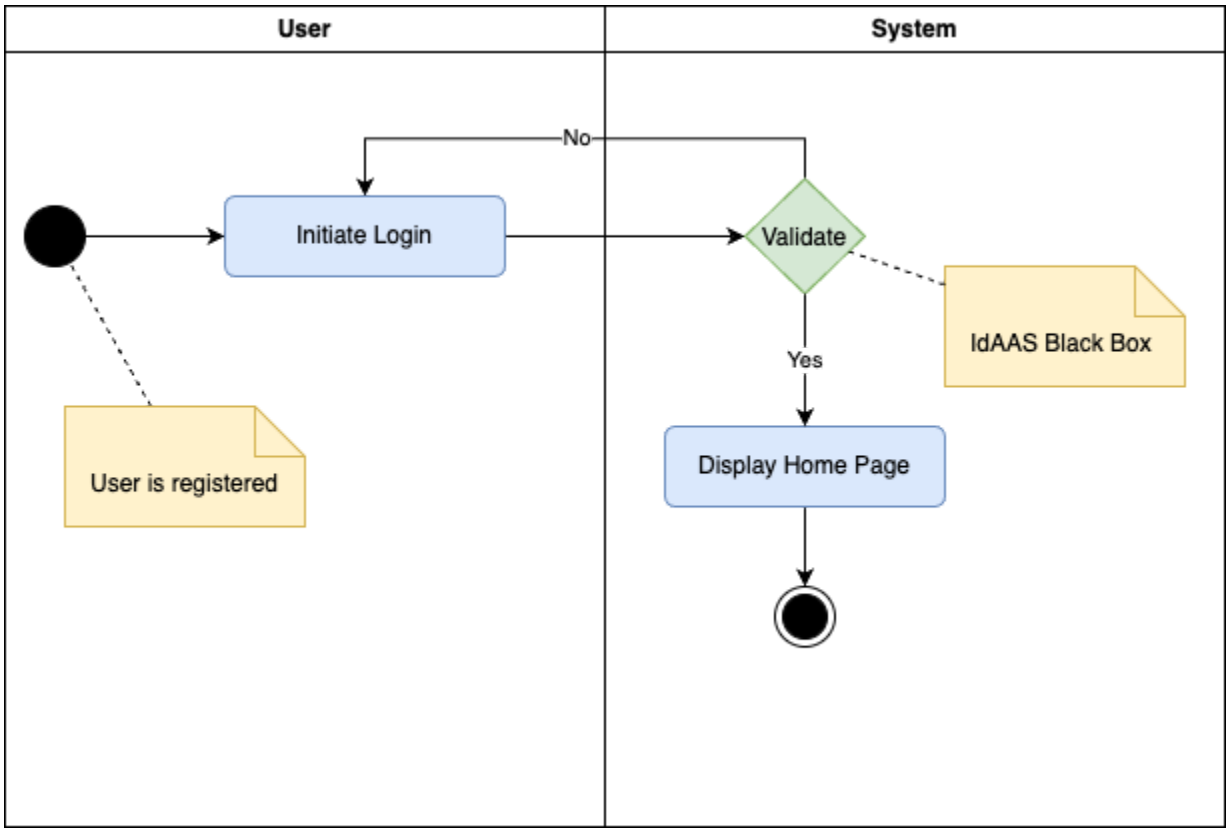
1. **Scalability:** CQRS can improve performance, scalability, and security because it allows read and write operations to be scaled separately.
2. **Simplified design:** By separating concerns, CQRS can make the code easier to understand, maintain, and evolve.
3. **Optimized data schemas:** The data store for reads (the query model) can be designed separately from the data store for writes (the command model), allowing each to be optimized for its specific needs.
4. **Better testing and debugging:** With commands and queries separated, unit testing can be more straightforward. Also, it is easier to replay commands for debugging purposes.

4. Requirement Realization

4.1. UC-08: Register to the Mobile Application



4.2. UC-01 & UC-09: Login Mobile and Web



The only difference between mobile and web logins is that the mobile login supports 3rd-party social providers. But since the authentication is handled by the IdAAS itself, and basically the process is a black box, we can depict the simple flow as the given activity diagram.

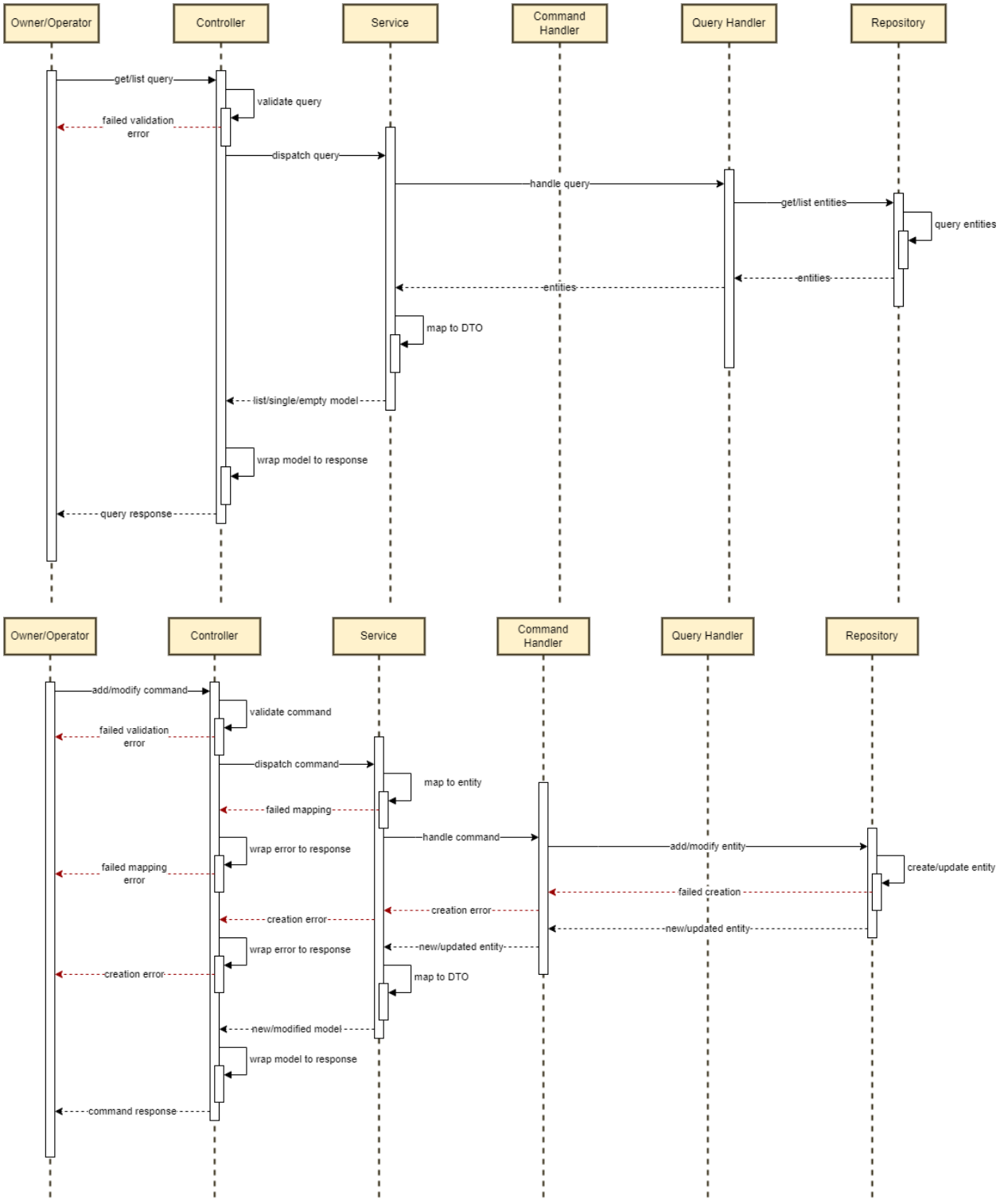
4.3. UC-02, UC-03, UC-04 and UC-05: Management

Though there are different use cases to manage operators, vending machines, products and inventories; since we apply CQRS by architectural pattern, the sequence models are similar.

In this pattern we have basically creation and modification flows and as stated in the patterns section, by separating concerns (queries versus commands), CQRS can make the code easier to understand, maintain, and evolve.

The query sequence is much straightforward without less failing points since it's a pure query and the worst case is an empty response model. On the other hand, the command sequence has multiple failing points like; validating failures, mapping failures, and creation failures.

A note on this sequence is that the model is transferred to entities in Data Transfer Objects - DTO and from models back to users in wrapped responses. Wrapping helps client applications to look at the result success code and any errors if any exist before trying to parse the response DTO.



The data model and class interactions can be tracked from the notebook.

4.3. UC-10: Find/Filter Nearest Machines

A medium-precision Geohash algorithm is used to identify the nearest machines in a 350 meters perimeter and filter them per request.

The logic is that the Geohash algorithm assigns a hash value for a given latitude and longitude. The end-result changes by the precision as well. We take the precision as 7 to fit our scale of searching nearby. The precision matches with the decimal precision of the given location coordinates.

```
Algorithm FindNearbyVendingMachines:

Input: UserLocation U, Set of VendingMachines V, GeohashPrecision P
Output: Set of VendingMachines S

Begin Algorithm:

1. Set h ← ParentGeohash(U, P)

2. Set S ← {}

3. For each v ∈ V:
    Set h_v ← ParentGeohash(Location(v), P)
    If h_v = h then
        Add v to S
    End If
End For each

4. Return S

End Algorithm
```

In this notation:

- ← is the assignment operator
- ∈ means "in" or "belongs to"
- {} denotes an empty set
- Geohash(Location(v), P) would compute the parent geohash of the location of a vending machine v at the given precision P.

4.3. UC-11: Rate Machines

Keeping the total count and the average of ratings, instead of individual ratings, can offer several advantages, particularly from a data storage, processing, and privacy perspective:

- Storage Efficiency: Storing only the total count and average of ratings can greatly reduce the amount of storage space needed, especially when dealing with a large number of ratings. This can be critical in large-scale systems where storage costs can become significant.
- Performance: When the average rating is frequently accessed (e.g., for display on a product page), it is more efficient to retrieve a single precomputed value rather than calculating it from potentially thousands or millions of individual ratings. This can also reduce the computational load on your servers and speed up response times.
- Privacy: In some cases, storing individual ratings may have privacy implications, as ratings can sometimes be tied back to individual users. Keeping only the aggregate data can help mitigate this risk.
- Simplicity: Keeping total count and average rating simplifies many operations. For example, updating the average when a new rating comes in is a straightforward operation of just updating two numbers.

For these reasons we realized this UC with average and total ratings. However, it's also worth noting the drawbacks of this approach. The most significant is that it loses all granularity of the data. Once we aggregate the data, we can't access or analyze the individual ratings anymore.

So we defined a decay ratio of 0.7 meaning older ratings would have less effect than new ratings and the average rating will be more sensitive to new ratings.

```
Algorithm UpdateRatingWithDecay:

Input: PreviousAverage A, PreviousCount C, NewRating R, DecayFactor D
Output: NewAverage A_new, NewCount C_new

Begin Algorithm:

1. Set A_decayed ← A * D

2. Set C_new ← C + 1

3. Set TotalRating ← A_decayed * C + R

4. Set A_new ← TotalRating / C_new

5. Return A_new, C_new

End Algorithm
```