**Low-Level Architecture and Data Models**

**<09>:<Petswala>**

**<team member names & ids>**

| **Student ID** | **Name** |
| --- | --- |
| **22100298** | **Roshan A. Aziz** |
| **22100201** | **Mohid Yousaf** |
| **22100111** | **Ayan Tabassum** |
| **22100004** | **Adil Aslam Chaudhry** |
| **22100248** | **Sabahat** |

**Table of Contents**

[1.](#_heading=h.gjdgxs) Introduction 3

[2.](#_heading=h.30j0zll) System Architecture 4

[2.1 Architecture Diagram—](#_heading=h.1fob9te)As it is in the prototype code 4

[2.2 Architecture Diagram—](#_heading=h.3znysh7)As it should-be 4

[3.](#_heading=h.2et92p0) Data Models 5

[4.](#_heading=h.tyjcwt) Tools and Technologies 6

[5.](#_heading=h.3dy6vkm) Who Did What? 7

[6.](#_heading=h.1t3h5sf) Review checklist 7

# Introduction

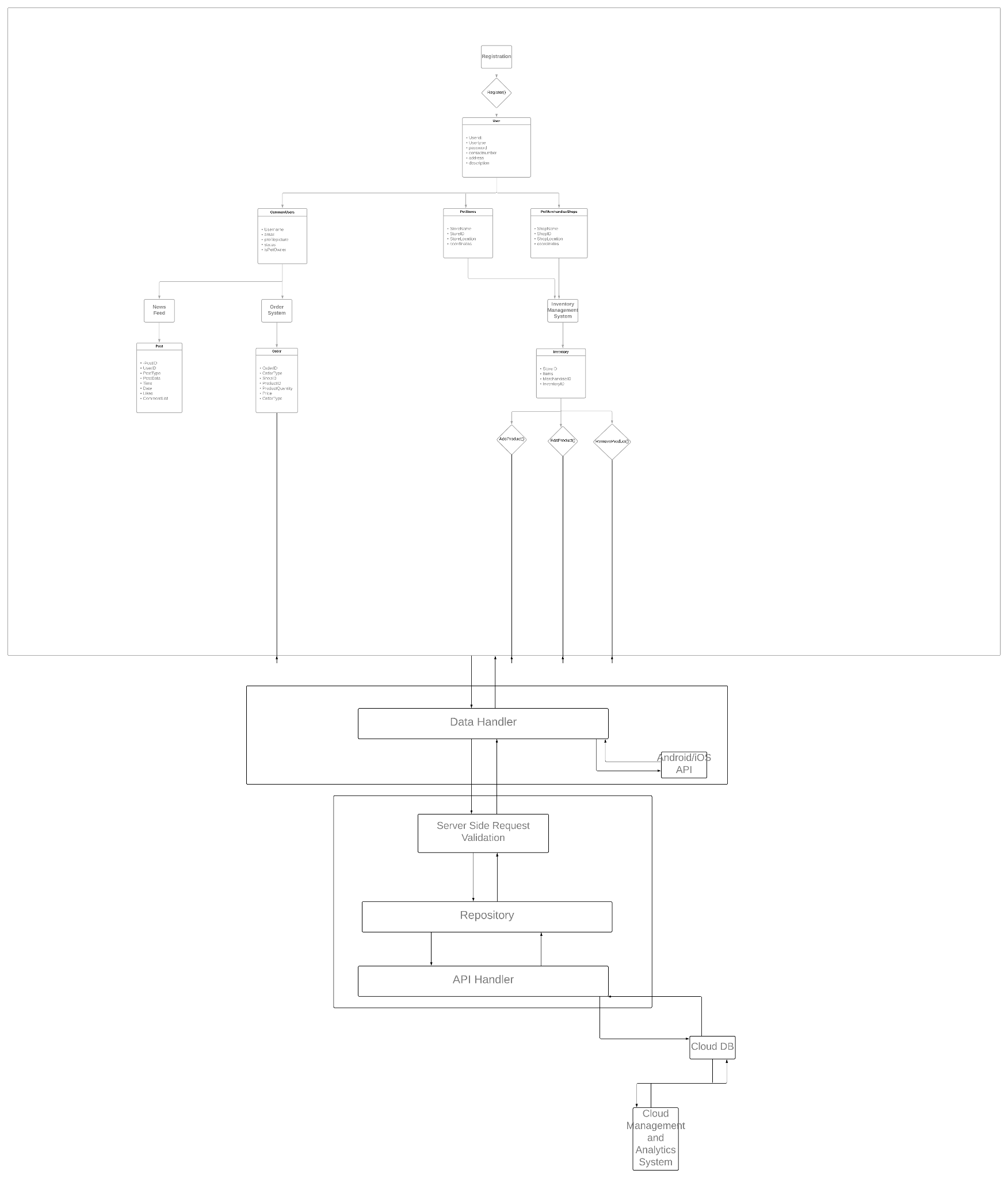
The project comprises an application catered towards pet owners, pet shops, sellers of pet accessories, veterinary doctors, pet rescue volunteers, and pet shelters/rescue teams. The application aims to link all these different entities through a variety of useful and helpful functionalities; pet owners would be able to search for different accessories, as well as finding good veterinary doctors for their pets, making their lives much more comfortable and better. They would also be able to interact with other pet owners and share photos and stories of their own pets as well. They could share helpful information with other pet owners. Any user of the app can also help with the animal rescue process by reporting pets to be rescued to the relevant rescue teams in the area.

# System Architecture

## Architecture Diagram—As it is in the prototype code

Due to the image compression of the diagram, we are attaching the link to view the full diagram in clarity.

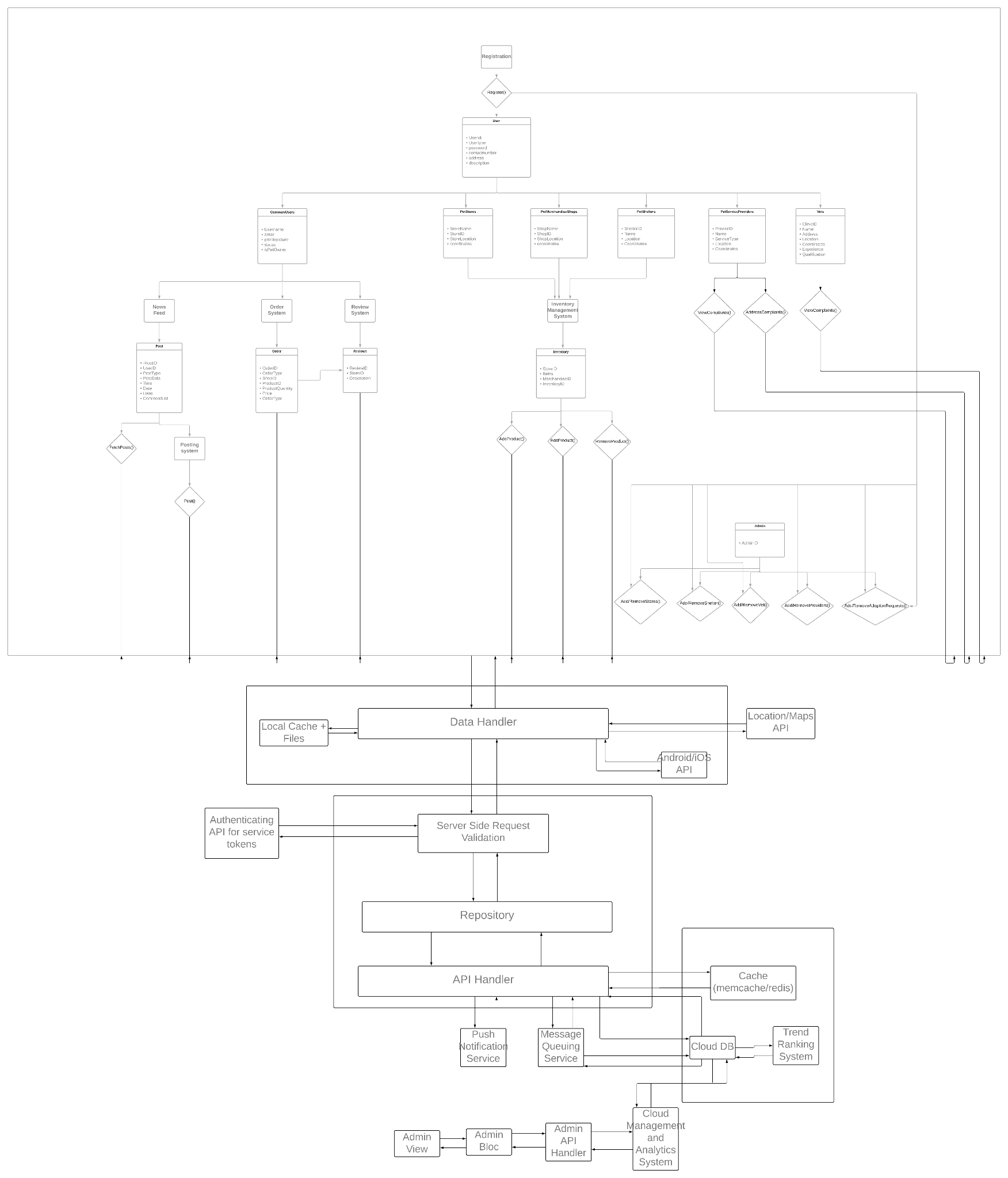
https://lucid.app/lucidchart/1fbeb4c9-12de-46f7-8ff0-8b1ccf4bc268/edit?invitationId=inv\_8477460e-3454-412d-9378-e36c3728d395



## Architecture Diagram—As it should-be

Due to the image compression of the diagram, we are attaching the link to view the full diagram in clarity.

https://lucid.app/lucidchart/39c78f76-1140-4cd0-ae84-54b67d239181/edit?invitationId=inv\_5aa8db70-3afc-484e-b241-a5bc3d6ea4ff



The updated architecture builds upon the previous architecture for the prototype by reusing the already built system and adding on to it. Our system uses the BloC Architecture keeping in mind the technologies we are using and the context of our system. For our prototype, the backend side of the system uses the core functionalities necessary for data management, such as, handling fetch and update requests for the PetStores, PetMerchandiseShops and CommonUsers classes. Now, the updated architecture reflects the scalability of the existing system by adding on the core functionalities, such as implementing local cache storage for authentication tokens sent by the Authentication API, to ensure privacy and security. There are many more APIs that perform different functionalities of smooth and efficient database management.

The business logic layer of the architecture contains the business logic of the subsystems present in a semi restricted layered structure, keeping the most used subsystems in the lowest layer, hence maintaining a useful layered structure. As our application is based on flutter, our application and business logic resides client side in order to ensure low-latency and a responsive UI design. The business logic is kept entirely separate from the UI and data layers to keep the system modular and scalable, both in terms of functionality and load. In order to communicate with the UI and repository layers, we use an event-based architecture with streams. Events trigger functions which perform business logic and state is updated via streams. This architecture is common and optimized for flutter and works well in separating state from UI.

For the purposes of demonstrating that, we will now assume the data flow of ‘fetchpost()’ function.

When the Common User wants to access the News Feed, the client-side sends a fetch request, more specifically, the ‘fetchpost()’ function, to the Data Handler API. The Data Handler first accesses the local cache to get the current authentication token stored and sends that as well as the fetch request to the Request Validation API. The API matches the token sent by the client with the one stored in the system for the particular user and if the token is not expired, the system sends an ‘error’ response back to the Data Handler. If it has not expired, the Request Validation API forwards the request to the Repository and the API handler. The repository layer is used to handle multiple sources of data from APIs. The API handler identifies the request and sends an asynchronous fetch request to the Cloud DB as well as the cloud-based cache simultaneously. The first response from either of the two requests will be catered to and the data flow will follow the same path but backwards. And since the front end data capture is built upon *Streams*, the data flow sent back will be automatically updated and reflected on the CommonUser user interface.

These systems in the architecture diagram are interconnected through data streams, they take data from each other. The user actions all take into account and depend on the preferences of the user. While the user actions all use the utility functions, which are the reusable components of the software, for various tasks and needs. We used cloud DB and storage options in order to ensure consistent latency and up-time and to ensure that there is not a single point of failure. Moreover, having scalable cloud storage helps in easy load scaling and it also facilitates a flexible load, so that costs are dependent on usage only. The system can also quickly adapt to very frequent changes in load and we will not have to physically adapt to such change in resource requirement.

Keeping the above in mind, our architecture has the following pros:

1. Easy to make changes in UI because of separation of state and UI
2. Easy to change data sources and APIs because dependence on only one layer
3. Scalable and flexible due to scalable cloud services and client-side logic
4. Client-side application logic to ensure low latency and increased responsiveness in UI
5. Data heavy tasks and logic delegated to a separate cloud backend to minimize heavy client-side computation.
6. Caching to ensure low latency and quick data delivery
7. Pre-computing heavy tasks ensures low run-time latency

However, there are a few cons:

1. Client side is thick as application logic resides there making the app larger.
2. Regularly computing trends is very resource intensive.
3. Client requests need to be re-validated server-side because we can never trust client-side data.

The classes in the architecture diagram:

**User** : It is a user of an application that can be subdivided into common users, Service Providers, Shelters , Veterinary Clinic, Pet Store and admin and it can register.

**Common Users:** Common users are those who are casual app owners and they can be pet owners, they have the functionalities of reporting stray animals, buy pets, finding veterinary options, post and review etc

**Pet Service Providers:** They are providing services related to pets, they can also update their services and address complaints if any.

**Pet Shelters:** Pet Shelters are entertaining rescue requests mainly.

**Pet Merchandise Shops:** Pet merchandise shops are the stores which sell products related to pets.

**Pet Stores:** Pet stores are the places where people can buy pets from. These are physical stores which are set up online on our app as well.

**Veterinary Clinic:** Veterinary clinic is a place where one or more than one veterinary doctors sit. These are also physical clinics with an online clinic set up on our app.

**Admin:** Admin is the owner of the app and they have special rights. They can add, remove people on the app, and modify several other things.

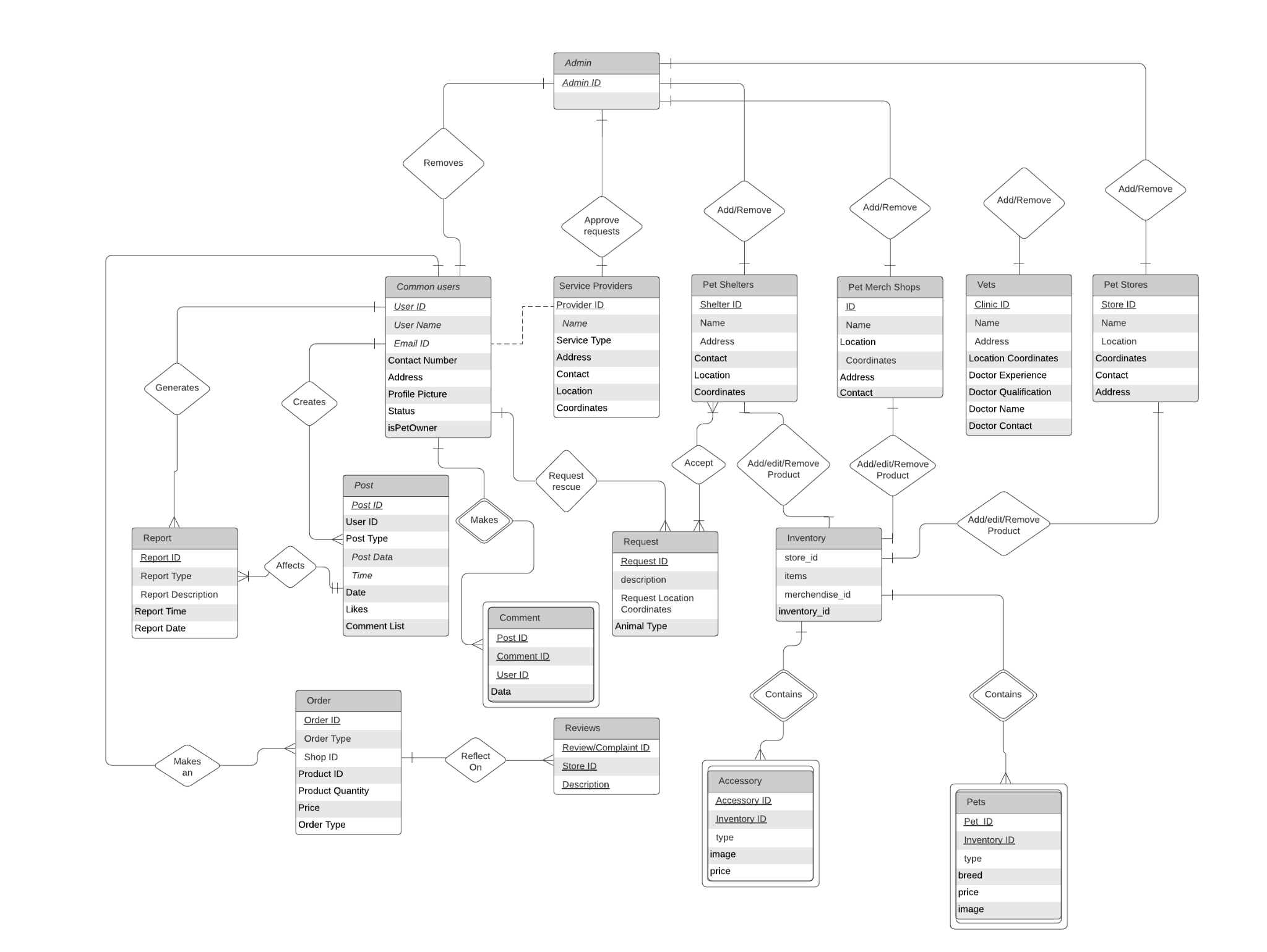
**Inventory:** An inventory is a list of items available at a particular store.

**Order:** A purchase is users buying the items in exchange for money. A purchase can be made for pets or accessories.

**Reviews / Complaints:** Reviews and complaints are the messages sent to store owners by buyers, to give feedback on their product or to register their complaints.

# Data Models

**ER Diagram:**



The data fields have been taken from the class diagram since they represent the variables or data that would be stored in the database. The entities chosen for the E/R diagram were also taken from the class diagram on the basis of them having or requiring a representation in the database. These entities include:

* Admin:

This contains the ids of the different admin to help them log in.

* Common users:

This entity contains all the information about the common user or pet owner type of user.

* Service Providers:

This entity contains all the information about the service providers.

* Pet Shelters:

This entity contains all the information about the Pet Shelters user type.

* Pet Merch Shops:

This entity contains all the information about the Pet Merchandise Shops.

* Vets:

This entity contains all the information about Veterinary doctors.

* Pet Stores:

This entity contains all the information about the Pet Stores.

* Report:

This entity is for the table that contains all the reports that have been generated on posts.

* Post:

This entity is for the table that contains all the data for posts, including the post data, the id of post and user, number of likes and comment list.

* Comment:

This entity is for the table that contains all the comments, this is a weak entity since it is dependent on the post.

* Request:

This entity is for the table that represents all the rescue requests for rescuing animals and contains the location of the animals and the type of animal.

* Inventory:

This is the entity that represents each shop, merch store and shelters and their goods, this entity contains ids for the different item type and the store.

* Accessory:

This entity has all the accessories data, basically, all the accessories in stock. This is a dependent entity and hence a weak entity. This is dependent on inventory.

* Pets:

This entity has all the Pets data, basically, all the Pets in stock. This is a dependent entity and hence a weak entity. This is dependent on inventory

* Order:

This entity represents all the transactions that happen on the application whether they be sales or purchases. This entity represents the table that stores the data for these transactions. It includes the order id, the order type(sales or purchase), quantity and price.

* Reviews:

This entity represents all tables containing the reviews left on the Pet stores, merchandise shops, vets etc. It contains the review id, the id of the store on which the review is left on and the review itself.

# Tools and Technologies

<List down tools and technologies that you are using for development and deployment. Make sure that you mention name and version of the tools.>

Flutter 2.0 for development on both iOS and Android

1. AWS Aurora for cloud DB
2. AWS elasticache for Caching servers
3. AWS EC2 for hosting and computation
4. AWS location API

# Who Did What?

| **Name of the Team Member** | **Tasks done** |
| --- | --- |
| Roshan A. Aziz | Worked on ER diagram & ER description. |
| Sabahat | ER diagram |
| Ayan Tabassum | Architecture Diagram and Description |
| Mohid Yousaf | ER, tools and tech, introduction, architecture diagram |
| Adil Aslam | architecture diagram, tools and tech |

# Review checklist

Before submission of this deliverable, the team must perform an internal review. Each team member will review one or more sections of the deliverable.

| **Section** **Title** | **Reviewer Name(s)** |
| --- | --- |
| Roshan A. Aziz |  |
| Ayan Tabassum | Architecture Diagram and Description |
| Mohid Yousaf | ER, Architecture Diagram, Tools and tech |
| Sabahat | ER diagram and description |
| Adil Aslam | Architecture Diagram, T |