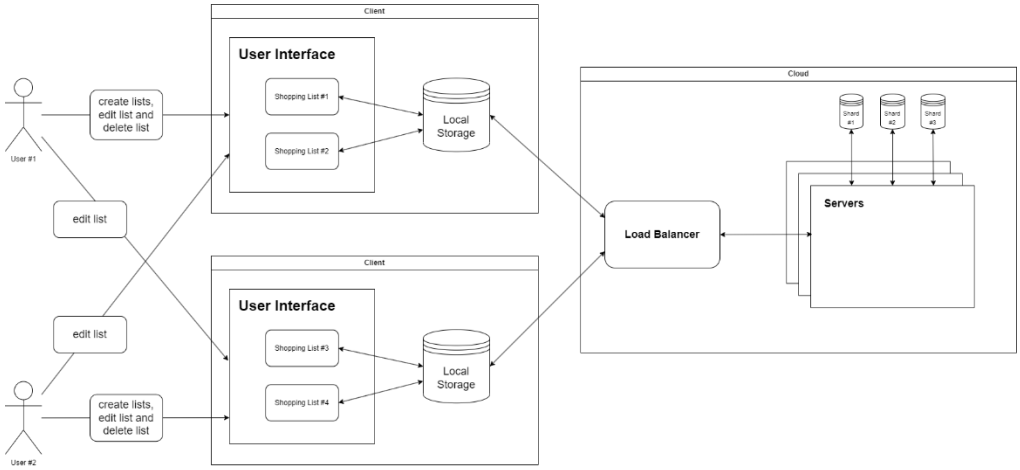
Shopping List Application

The main objective of this project is to create a local-first shopping list platform, that is designed to provide users with the ability to create, share and modify shopping lists, while ensuring data persistence and high availability.

General Architecture

The system is composed of two main components: the client side, which is running on the users' devices, and the cloud side, for backup storage and data sharing.
With this in mind, we designed a platform architecture that takes this into account.



Client Side

The client side of the application provides a user-friendly interface for managing shopping lists. Users can create, edit, and delete their own shopping lists, as well as access and edit shared lists with a unique identifier (UID).

Associated with each item is its available stock and the user's desired quantity. When an item's stock becomes 0, the item turns unavailable, until another user removes it from their shopping list. This item stock will be kept in a counter that stores the remaining available stock of the respective item.

The application utilizes local storage to store and manage shopping list data (with the purpose of only viewing the list) and quick retrieval of information. Local storage is synchronized with cloud storage to ensure data consistency across devices and maintain real-time updates for shared lists.

Cloud Side

Regarding the cloud side of the platform, it serves as the central repository for all shopping lists data, ensuring data persistence and synchronization across devices. It also allows the sharing of shopping lists between users and provides data backup and recovery in case of data corruption or a device breaking down. On another note, this side will also be responsible for storing all the information regarding each product and its available quantity.

As we are aiming for millions of users, we need to guarantee that data access bottlenecks are avoided. Therefore, we designed a cloud-side architecture that has the following features:

Data Sharding

Considering a scenario where millions of users are using our shopping list platform. If all the shopping lists were stored on a single database, it would become overloaded as the number of lists and concurrent requests continue to grow. Therefore, to battle this we can shard the data into smaller, self-contained segments (shards), that will be allocated across multiple database servers.
This way, we'll achieve high scalability and a boost in query performance, due to the parallelism that results from this data division.

Data Redundancy

To ensure high availability in our application we need to deal with failures. One way to tackle this problem is by having multiple instances of the same data. This way whenever a node fails there is at least one another that can satisfy the user's needs.

Load Balancing

As the number of users and concurrent requests on our platform, the load on the application servers can become overwhelming. To handle this increasing demand and ensure a responsive and scalable platform, we believe that we need to implement a load balancer.

This load balancer will be responsible for handling the traffic routing, by distributing incoming requests across multiple application servers. If all connections require roughly the same computing power, we believe that a Least Connection Algorithm would be the most appropriate for this scenario.

CRDTs

Shopping List & Stock

Data Structure: In this context we have two similar situations: a shopping list made up of many products that have an id and the desired quantity, and a list of all products with their id and their remaining quantity. Therefore, we can think of these lists as maps (key-value pair) where the key corresponds to the product's id and the value is its quantity. With this in mind, we can use **Map CRDTs** which are a powerful tool for managing maps in distributed systems. In this case, since we want to assure the addition, removal, and update of a shopping list's items between different users, there needs to be the synchronization of the key-value pairs.
Concurrency Semantic: Last-Writer-Wins.
Synchronization Models: Operation-based and State-based.

Product

Data Structure: Each product quantity in the lists can also be perceived as an independent CRDT. Therefore, to guarantee synchronization when different users update the desired quantity of the same product, on a shopping list, we consider the **PNCounter (Positive-Negative Counter)** to be the most adequate type of CRDT.
Concurrency Semantic: Last-Writer-Wins.
Synchronization Models: Operation-based and State-based.

Chosen Technologies

To implement this design, we have decided to use the following technologies:

Client Side

- **React** - for building an interactive user interface.
- **IndexedDB** - for storing larger amounts of structured data with advanced indexing and querying capabilities.

Cloud Side

- **MongoDB** - besides its good performance and scalability, due to its horizontal scalability, ability to deal with unstructured data and ease of sharding implementation (already implemented in its core architecture) we think that MongoDB is better suited for this project, for storing data.
- **NGINX** - to act as our web server, for the cloud side of the application. We chose it, not only due to its popularity, but also because it can easily act as a load balancer and reverse proxy to improve resource availability and efficiency.