SDLE Project - Shopping List

Overview

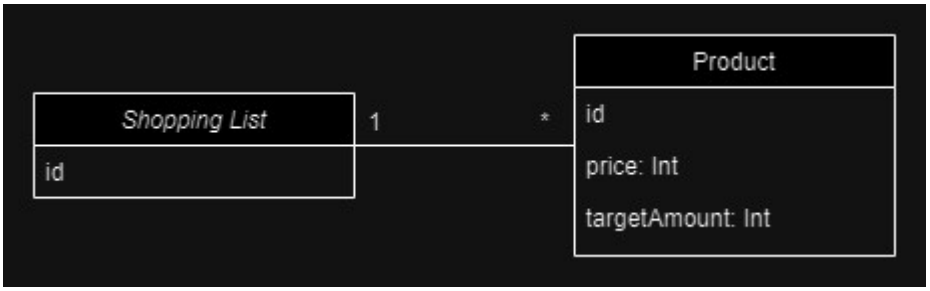
Within the scope of the SDLE discipline, the creation of a local-first shopping list application was designed. The requested app should persist data both locally and in the cloud.

It should allow users to create new lists or access already created ones, as long as they know the list ID, being able to add or remove products from the lists. Development must also have a strong focus on high data availability and take into account the possibility of millions of simultaneous users.

Technologies

- Frontend: Python with Flask
- Backend: Python with Flask_sqlalchemy

Data Model



The data model is defined between two classes, Shopping list and product.

Shopping List:

- id - Unique identifier, used as key for retrieval and as “URL” to be selected by users to be viewed and edited.

Product:

- id - Unique identifier, used as key for retrieval
- price - Price of each element of the product
- targetAmount - Number of elements of that product the user wants to buy

There is no need for a client/user class, since every user with the shopping list “URL” can edit and view it like its own.

Application Architecture

The application structure is composed of three levels, client-side, middleware, server-side.

The client-side is divided between an interface part, responsible for interaction with the user, allowing the creation, viewing and editing of lists, and another responsible for data persistence locally and synchronization with the server.

The interface consists of an index page, where multiple shopping lists are visible; There is a simple form to add a new empty list. Selecting a list takes the user to a new page, where it shows the lists' items (name, quantity and price), as well as a form to add new items. There is also an option to clear the whole list of its items.

Back-end and Theoretical Concepts

In this project we aim to create a local-first shopping list with high data availability. With that aim in mind, we decided to take inspiration from the model presented in "Dynamo: Amazon's Highly Available Key-Value Store".

In order to mimic the Dynamo system, we'll be using principles such as Conflict-Free Data Types (CRDTs) in order to achieve eventual consistency, Consistent Hashing and data replication, in order to avoid losses even if a server goes down.

To be more specific, we will implement Delta CRDTs, as we aim to make small but frequent changes to the objects, ensuring a local-first environment. This entails making the objects oriented towards update operations (addition and subtraction of integers in our case), which the client side will be providing to the server. The server will then instantiate and propagate every Delta (change) to other relevant clients and servers. This means that if two clients ever have different states for the same object, their changes will just be propagated to the server, who will solve the conflict by instantiating every change made and updating the clients. This also means that if a client is ever disconnected, as soon as connection is restored information will be updated on all sides.

By doing this, we enable users to make simultaneous changes to shopping lists while ensuring that these changes can be reconciled seamlessly. After an analysis of the credit catalog, it was decided that a implementation of two delta-state CRDT's was necessary:

- Counter - for the targetAmount of a product element;
- Dictionary/Map - each list as a dictionary, linking each item id to the targetAmount

Client-Side

The client side will be the most simple component, containing the implementation for the CRDTs and maintaining communication with the server-side through the middleware. It will register the creation of new objects, as well as any changes to the Counter CRDT as a delta, and send those to the middleware so that it distributes it throughout the servers. It will also be able to request data though the list's IDs, after which it will be constantly reading for updates from the server.

Server-side

The server-side component is responsible for handling requests from the client-side component, managing the database, and ensuring high availability of shopping lists. Each server instance will contain a similar amount of data, provided by the consistent hashing method.

Here we use a middleware as a Coordinator for where to store each piece of information. The Coordinator will have a copy of all keys of all objects and servers, coordinating the storage of data through the servers in a distributed manner, as well as safety copies in case of a server failure.

This will be done through Consistent Hashing, meaning that a hash function will be used in every server name and object in order to place them in a set ring of values which correspond to data-space, this model will then be used to instantiate the data distribution.

Assuming we want N copies for safety for each piece of information, in case of a server failure, the next N-1 servers in the ring will also contain that data, reducing the complexity needed for readjusting the system and moving data to new servers.

New server additions will also be made easy by this model, as it only requires moving the data around the new server node in the ring.

The Servers themselves will be treated as nodes in the previously mentioned ring, and will communicate with the coordinator, handling conflicts by instantiating received deltas and propagating said changes back to the clients.