Eliza Kraule emk6 elizaMkraule
Audrey Pizzolato aap13 audreypizzolato
Sebastian DiPrampero sjd7 sebastiandiprampero

Description of Design:

Upon OwlDB initialization, the system is started by the package databse_host which encapsulates the storage of databases in the system. When a client sends an HTTP request it gets delegated to the handler package which first authorizes the request by forwarding the request to the authorize package. If the request is authorized the handler package interprets the HTTP method and forwards the request URL to the parser package. The parser package parses the request URL and returns it to the handler to validate the parsed path in the database according to the HTTP method. Assuming it's a valid request either database_host, database or docAndColl package finishes the respective request on the corresponding object in the system. Package database encapsulates the storage, retrieval, manipulation and deletion of databases and documents in databases. Package database_host interacts with the package database to store multiple databases in the system. Package docAndCol handles manipulation, retrieval, insertion and deletion of documents and collections.

Here is a more detailed description of how the different method requests interact with packages, note that for all method requests the handler calls the parser package to parse the path and the handler validates the parse.

Given a “PUT” request, the handler determines which structure is being inserted and calls the ‘parent’ object to insert the structure. All of the insertions are executed within the database_host, database, document or collection file. Upon a document insertion request the package validator validates the document body against a given json schema.

Given a “GET” request we also check if the client would like to subscribe to a document or a collection, which then are notified if there are changes to the given document or collection. Creation of subscribers and subscriber notification is encapsulated in the docAndColl package.

Given a “PATCH” request a document is patched by the jsonPatch package which uses the jsonvisit package containing the jsonVisitor design pattern. After patching the final document is validated by the validator package against the given json schema.

Given a “POST” request, we can either authenticate a user which is done by the authorize package or create a new resource which is done by the

Given a “DELETE” request the handler determines which object is being deleted and calls the ‘parent’ object to delete the object. All of the deletes are executed within the database_host, database, document or collection file. A “DELETE” request may also be used to remove an authenticated user which is done by the authorize package.

We decided to maintain the database and collection as two separate structs and packages for multiple reasons. The first deals with subscriptions, collections allow users to subscribe to packages however databases do not have this feature. In addition, there is a cyclical nature between documents and collections that does not apply to databases and documents. Finally, there is better readability with having a separate database and collection package.

If we had more time on the project we would have created more interfaces for database, document and collection maintenance as well as possibly splitting document and collection into different packages without creating an import cycle.

Design Principles

Single Responsibility Principle is a design principle that emphasizes cohesion between abstractions. The primary design of our program had a function handler that acted as the main point of interaction. To ensure clarity and separation, the handler did not take on multiple tasks. Instead, it delegated specific responsibilities to specialized components. For instance, when it came to deciphering the URL, a dedicated parser was called. This enabled us to isolate potential issues related to URL parsing and streamline testing. Similarly, the tasks associated with database operations, document management, and handling collections were segmented into distinct packages. By compartmentalizing these functionalities, not only did we enhance the modularity and

maintainability of our codebase, but we also ensured that each module or package had only one reason to change.

Liskov Substitution Principle is a design principle that focuses on maintaining the expected behavior of methods otherwise the function will not work correctly. Our group put an emphasis on breaking down different functions such that they executed only one task. Not only did this allow our functions to be reused, but it allowed clarity in testing and design. We split our parser into multiple different functions that could be used to parse URL for all of the different request methods. We had a parser function that split the URL into different segments and then a different validator of the URL for put and get requests. In addition we had functions to get a document, collection or database as a pointer and a separate set of functions to format the document, collection or database to the response writer. This enables us to reuse the function to get a document, collection or database multiple times. In addition it makes each of the functions extremely clear.

Dependency Inversion Principle hones in on minimizing or eliminating dependencies between entities. As a design decision, our handler is the point at which all of the dependencies lie. Our handler is the point of connection between the parser, validator, authorize and calling the insert, delete, and find requests for database_host, databases, documents and collections. Having the handler as the point of dependencies allows all of our other functions to be independent of each other allowing for isolated tasks to be completed as well as testing independently. One other point of dependency that was difficult to avoid was between our database, document and collections structure. This is because database struct holds a skiplist of documents, documents holds a skiplist of collections and collections hold a skiplist of documents. Because there is an inherent dependency between the structures they all are dependent on each other. To mitigate some of this dependency when designing the program, our skiplists can take different type parameters such that they are able to be reused in all of the different structs, so we only need to build and test one skiplist and it is not specific to a struct.

Concurrency

All of the concurrency is managed in the skiplists and the handler requests. Handler requests can be called concurrently, then the concurrency is handled in the skip lists in order to prevent data races and maintain safety and liveness. The structure of the skiplists are as follows: Database has a skiplist field that contains all of the documents in that database, Documents has a skiplist field which contains all of the collections in the document, and Collections has a skiplist field that contains all of the documents in that collection. When finding, inserting or deleting a document or collection in a skiplist, there is an issue of maintaining safety and liveness during concurrent requests. So our skiplists are constructed to manage concurrency.

The skiplist design uses mutexes and atomics in order to successfully execute concurrent requests in a safe way. The skiplist is made of nodes, which each contain a mutex, a marked flag which signifies that the node will eventually be removed, and a fully linked flag that indicates whether a node has completed the insertion process. The skiplist has the four methods Find, Upsert, Remove, and Query, which each use atomics but vary on the other fields used from the nodes. The Find method is able to run concurrently using just the fully linked and marked flags, since it is only reporting whether the node was found in the list. The Upsert and Remove functions need to use locks in addition to the flags since they can actively change the skiplist's contents which can affect other operations dealing with the skiplist concurrently. The Query method works similarly to the Find method in that it does not need to use locks, but it does need to ensure that after it establishes a list of nodes, this list has not been changed in the time it took to run the method. To do this, it uses a timestamp that increments after inserting, updating, or removing a node. It constructs a list of nodes twice, and if the list remained the same and the timestamp did not change, then we know that this list is concurrently safe.

Architectural Diagram of our system.