P1Group70 Design Document

- Joonha Kim jk67 jhkim00810
- Michael Li zl116 ZeningLi0402
- Rex Le rl105 RexLe192010

Design Principles:

Single Responsibility Principle: Our functions are clearly named and were designed to do one thing and do it well. For example, we design functions with clear usage and simple, direct names like “getdoc” and “ getcol”. For more complicated cases, We also offer descriptive names like” Parsepath”, which have a single responsibility to parse the path and lead them to the correct handler to handle it. In addition, Each module is designed with a distinct, well-defined responsibility. It contains only the functions, structs, and supporting helpers necessary for its specific functionality. For instance, the paths module focuses solely on processing pathname strings from requests, and the document module exclusively manages the creation, manipulation, and retrieval of document data. We also have related test modules like “paths_test” and ”document_test” that are only designed to test the correctness of the correlated modules.

Open/Closed Principle: The owldb database adheres to the Open/Closed Principle through the use of three core interfaces: IDocument, ICollection, and ICollectionHolder. These interfaces represent documents, collections, and collectionholders, allowing the database to reference these abstractions rather than their concrete implementations. As a result, clients can extend or replace these data structures with their own implementations without modifying the original source code, promoting flexibility and maintainability.

Interface Segregation Principle: Owldb's architecture is composed of three main elements: documents, collections that store documents, and collectionholders that store collections. These elements are represented by corresponding interfaces that define only the essential operations—such as put/get methods and other HTTP handler functions. More specialized functionality, like subscription management, document patching, and overwriting, is separated into distinct interfaces (Subscribable, Patchable, and Overwritable). This design ensures that clients are not burdened with implementing features they don't need, offering greater flexibility and customization in their implementations.

Concurrency :
Concurrency is managed in packages such as Skiplist, authentication, and Subscribe. And here are the details about how our concurrency works.

Authentication:

The authentication system is built around an Authentication struct that manages user sessions, using sync.Map to enable thread-safe concurrent access. This allows multiple goroutines to read and write session data simultaneously without introducing race conditions. Methods like ValidateToken, login, and logout safely interact with the session data through sync.Map operations such as Load, Delete, and Store. Since session data is immutable after creation, it improves concurrency safety. Cryptographically secure tokens are generated in a stateless manner, ensuring safety for concurrent usage.

Skiplist:

We utilize skip lists for fast storage and retrieval of data in our system based on names, particularly for documents and collections. Each node in the skip list has a mutual exclusion lock to handle concurrent modifications. Additionally, both the nodes and the root structure use atomic booleans and integers to track changes without locking, allowing for concurrent reads without the need for synchronization.

Subscribe:

In our subscription system, we use sync.Mutex and sync.RWMutex to handle concurrency and ensure thread-safe operations. Each Subscriber has a sync.Mutex to prevent race conditions when sending updates, closing the connection, or accessing shared resources like the Closed status. In SubscriberManager, we use a sync.RWMutex allows multiple goroutines to read the subscriber list concurrently, while writes (such as adding or removing subscribers) are serialized with a write lock. Updates are sent to subscribers via a buffered channel, and timeouts prevent blocking if a subscriber is slow to process updates. This approach enables concurrent reads and writes, ensuring efficient and safe handling of multiple subscribers and updates simultaneously.

Architectural Diagram

This design diagram illustrates the interactions between the components of owlDB, a system designed to handle REST API requests from external software. The Request handler dispatches these requests to appropriate handlers.Authentication is managed by an authentication handler that maintains a map of login tokens to user credentials and their expiration times. This handler is embedded in the DB request handler and validates tokens for incoming requests.The DB request handler processes requests by using the path processor to locate the correct resource, then invoking methods from either the collection or document interfaces. Both collections and documents implement interfaces for interacting with each other, contain skiplists, and maintain lists of subscribers. These components are responsible for processing user requests, writing responses, and managing subscribers.Error messages are handled by the error response method, which formats them for the client. This streamlined architecture ensures efficient handling of user requests, authentication, and error responses across all components.