COMP 318 Group 35 Design Document

Members, netids, Github ids:
Kofi Amoo, kda5, @headphoenix
Andy Lee, al116, @andyhlee49
Sunny Sun, ss303, @ss303

Design Principles:

The first design principle we focus on is the single-responsibility principle. This principle emphasizes cohesion, meaning that all abstractions should be cohesive, doing one thing very well and completely. Additionally, each abstraction should be well named. Our program decouples different aspects of the project into separate, well-named packages, ensuring that each one handles a distinct task. The benefits of this approach, which we experienced throughout the project's development, include improved testability and scalability as we progressed from checkpoint one to our final design. In the storage package, we focus on data-related operations. We have separate files for databases, collections, and documents, as well as individual files for the storage tree and node. In the storage package, we can store, update, retrieve, and delete documents and collections. Importantly, our storage package does not handle HTTP request reception or interact with authentication mechanisms. This design choice aligns with the single-responsibility principle, as we wanted data and metadata handling, as well as storage, to be separate from other external systems, such as HTTP request handling and authentication. On the other hand, our handler package, which consists of two clearly named files (authentication.go and handlers.go), is responsible for the reception and processing of incoming HTTP requests, including Server-Sent Events. It processes these requests, handles authentication and validation, and then passes them down to our storage system. Similarly, our skip list and subscription functionalities are implemented in separate packages, allowing each to be independently managed. At a lower level, in both of our main packages (storage and handlers), each file is responsible for a specific part of the project and works cohesively with the others to make the project function effectively. This level of modularity, decoupling, and cohesion contributes to the project's extensibility and maintainability, as discussed in class. Both storage and handlers handle their specific responsibilities without needing to know the inner workings of the other. Additionally, all files follow a consistent naming convention, ensuring that their names clearly reflect their respective functionalities.

The second design principle we employed is the Dependency Inversion Principle. This principle aims to minimize and eliminate dependencies between entities, as discussed in class. In our code, we apply this principle primarily through the use of interfaces in our system architecture. These interfaces help minimize and eliminate dependencies effectively. A key design choice was to keep these interfaces small, ensuring that each one adheres to a single responsibility, in line with eliminating dependencies. Our flusher, IStorage, IChildNode, and RequestPack interfaces are all designed following this principle. Our RequestPack interface handles different types of requests. Doing this abstract interface design rather than concrete implementation makes it so that multiple similar functionalities can be abstracted without the need for change in the code, as it makes it overall more flexible. For example, our RequestPack decouples the storage system and child node logic from the exact implementation of the specific request. Because we have this interface, Storage and other high-level components only need to know how to interact with the interface. This approach not only makes the code more flexible but also aligns with the single-responsibility principle, as each interface is focused on doing one thing very well.

The last design principle we use is the open-closed principle. This principle focuses on enabling extension without the need for modification. We applied this principle primarily by designing our system in a way that abstract away central functionalities, allowing new functionalities to be added with minimal changes to the existing code without modifying its core components. For instance, our design of the SkipList module provides a flexible indexing mechanism that can be extended or replaced without altering other parts of the system that rely on indexing in the future. Additionally, our handler package and storage package are designed with well-defined structures that separate the implementation of methods to handle different requests, making it possible to add new request types or storage strategies without modifying the core logic. Our adherence to the open-closed principle makes our project more maintainable and adaptable to evolving requirements.

Concurrency Management:

We manage concurrency in multiple ways. The first is the usage of mutex locks to ensure object safety and prevent double access. This is particularly important in our skip list, where each node has its own mutex lock. Whenever an operation modifies a node, it specifically locks the node used in the operation. For example, during an upsert operation, only the node involved in the upsert is locked, allowing concurrent access to other nodes while ensuring that the locked node is not modified by another operation. This concurrent skip list design minimizes wait times and is more efficient than using a single lock for the entire data structure. Go channels are also used for SSE and to handle certain aspects of authentication. Mutex locks prevent race conditions and ensure that data is shared in a synchronized manner even when multiple threads are modifying it. Go channels are particularly useful for enabling message delivery between the client and server without interfering with other operations. Overall, our database ensures concurrency and thread-safe operations, combining mutex locks for critical sections and Go channels for efficient communication, thus providing a robust solution for managing concurrent tasks.

Architectural design description:

As shown in the flowchart below, the backbone of our OwlDB project relies on a tree-like storage structure implemented in the storage package, supported by the childNode interface. The storage package consists of multiple components, including rootNode, database, document, and collections, which together form the tree-like structure for managing stored data. The main file calls the owldbHandler to manage incoming HTTP requests, routing them to the handlers package, which consists of authentication and handlers modules for processing requests and managing authentication. The handlers package interacts with the storage package to process data-related requests, while at the same time integrates with the subscription package to handle real-time SSE, ensuring subscribers are notified of updates. The skiplist module provides efficient indexing of nodes for the storage package, enabling fast data operations such as upserts, queries, and deletes. The jsonvalue module in the jsondata package is used to convert schema files into JSON validators, which helps ensure that response bodies are constructed correctly in the storage package, maintaining modularity and cohesion throughout the system.

Architectural Flowchart