Stefan Boskovic	sb121	sboskovic1
Cameron Liu	hl101	Cami101
Leo Li	jl257	JLLeo

OwlDB Design

To achieve our goal of implementing OwlDB in the most efficient way we could think of, we designed a package structure where 10 distinct packages plus main worked together to implement every distinct component of the database. Since we had the freedom to create packages to achieve any goal, we decided to use this to our advantage to keep main as simple as possible. Main only imports one package, the system package. We changed very little of main, which included parsing the inputs and creating a new system object. The system package was build to represent the top level of the database, handling http requests and using the top level database list defined within to delegate to other packages. This is the first example of how we achieved concurrency. The system package acted as our server, meaning it had to be able to handle concurrent requests and use a mux to separately handle each request.

The system package has many imports, mostly for the purpose of allowing it to delegate to other packages. This includes subscription, which must be called for most http requests to see if any subscribers should be notified, as well as authentication which also must be called before almost every request to see if the user is properly authenticated. Authentication also works to initially authenticate the user with a bearer token. The system package also imports the fileJson package. This package represents an interface, containing the methods Get, Put, Delete, and Next. This interface allows for us to minimize dependencies and make our functions more general, as all of these functions stated above are called on fileJson structs, which can be any of Collections, Documents, or the System.

This interface is an example of the dependency inversion principle. We would have to generically use collections, documents, and the system throughout our packages, which could create a nightmare of dependencies. We used the fileJson interface so all packages besides the system and one of document or collection could use the generic version of these files without an actual import. This allowed each of these packages to delegate the function to be called at runtime based on which file was passed, leading to fewer imports and dependencies. Lastly, system did need to import document and collection, the packages containing structs representing Documents and Collections. We needed to use the specific methods of these packages, but still tried to use the generic fileJson as often as we could.

The last two packages imported by system are skiplist and validation, and these two packages showcase the open-closed principle. The skiplist packages consisted of the concurrent data structure, skip lists, and the concurrent methods used to access and modify them. Skip lists are used as the means of storage for documents, collections, and the top level system. Validation would instantiate a validator and use it to verify any json object against a schema passed as a field to the validator. Both of these packages were built to be as generalizable and reusable as possible in accordance with the open-closed principle.

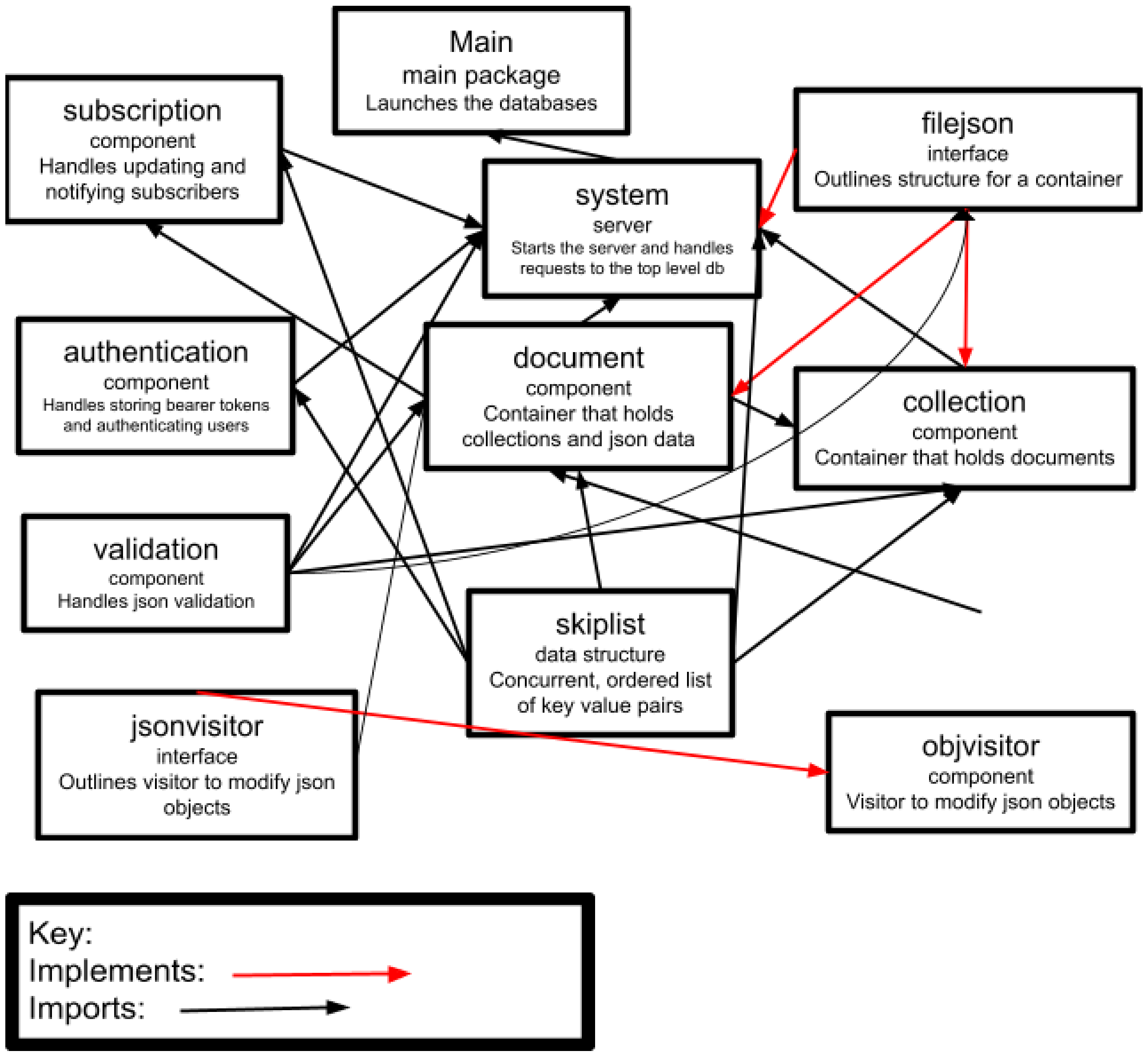
Skip lists were build using the most generic possible types, which was cmp.Ordered for the keys and any for the values. This allowed us to use skip lists for all fileJson types as well as for storing channels and subscribers as a part of the subscription package. Skiplists can also be extended to any other use if we ever needed to update the database in the future. Validation follows the same principle, where validator structs can be instantiated with a given schema and can be used anywhere for any kind of json validation. These structs can take in any schema required to check against and can check any json object passed.

Our last two packages were jsonvisitor and objvisitor, which both worked to modify existing json objects. Jsonvisitor was a provided package that contained an interface we were intended to implement as a part of a visitor struct. This struct was implemented in objvisitor, which used the accept function to modify any existing json object in accordance with the instructions passed as parameters.

Our package structure was designed to follow the single responsibility principle. Each package defined has one job and/or represents one higher level struct (with some children as well possibly). Every package defined is designed around one task and defined main and helper functions that only operate on its own structs or work to complete its own task. This helped us decouple our code, as every package only really operated on instances of itself. If a struct from a package had to be used elsewhere an instance of that struct would be injected and would be able to call its own methods defined in its respective package.

The other main way we achieve concurrency in our database is through the skiplist package. As stated, skiplist defined a concurrent, ordered data structure of key value pairs which we use to make all packages that require some form of storage concurrent. Skiplists are used in the system, documents, collections, authentication, and subscriptions. They are used in the first three to access and modify json documents or collections, and in the last two to concurrently store and access data about subscribers and authenticated users coming from http requests. Skip lists achieve concurrency by using atomics, locks, and checks to access and change data. All of the methods implemented in skip lists are fully concurrent, where they lock nodes that are currently being inserted, deleted, or modified, and have checks alongside this locking to make sure that no other goroutine is currently accessing these nodes. These checks alongside every operation that has to be concurrent using atomics allows our skip lists to be fully concurrent.

The final dependencies we have within our design are within documents and collections. Documents and collections have the method calls associated with http requests within their respective packages. This means that each of these packages need their own skip lists within their respective struct to store data regarding their child files, requiring them to both import skiplists. They also both need to validate json data being passed through put or patch, meaning they both need to import validation as they will be using an instance of the validator injected by the system package to check json inputs. Lastly, since the document package defines patch, it also needs to implement objvisitor and the jsonvisitor interface. Patch requires json data to be modified, which can only be properly done in our code using the visitor pattern within the objvisitor package.


Main
main package
Launches the databases
subscription
component
Handles updating and notifying subscribers
system
server
Starts the server and handles requests to the top level db
filejson
interface
Outlines structure for a container
authentication
component
Handles storing bearer tokens and authenticating users
document
component
Container that holds collections and json data
collection
component
Container that holds documents
validation
component
Handles json validation
skiplist
data structure
Concurrent, ordered list of key value pairs
jsonvisitor
interface
Outlines visitor to modify json objects
objvisitor
component
Visitor to modify json objects
Key:
Implements:
Imports: