Decentralized Cluster-Based NoSQL DB System

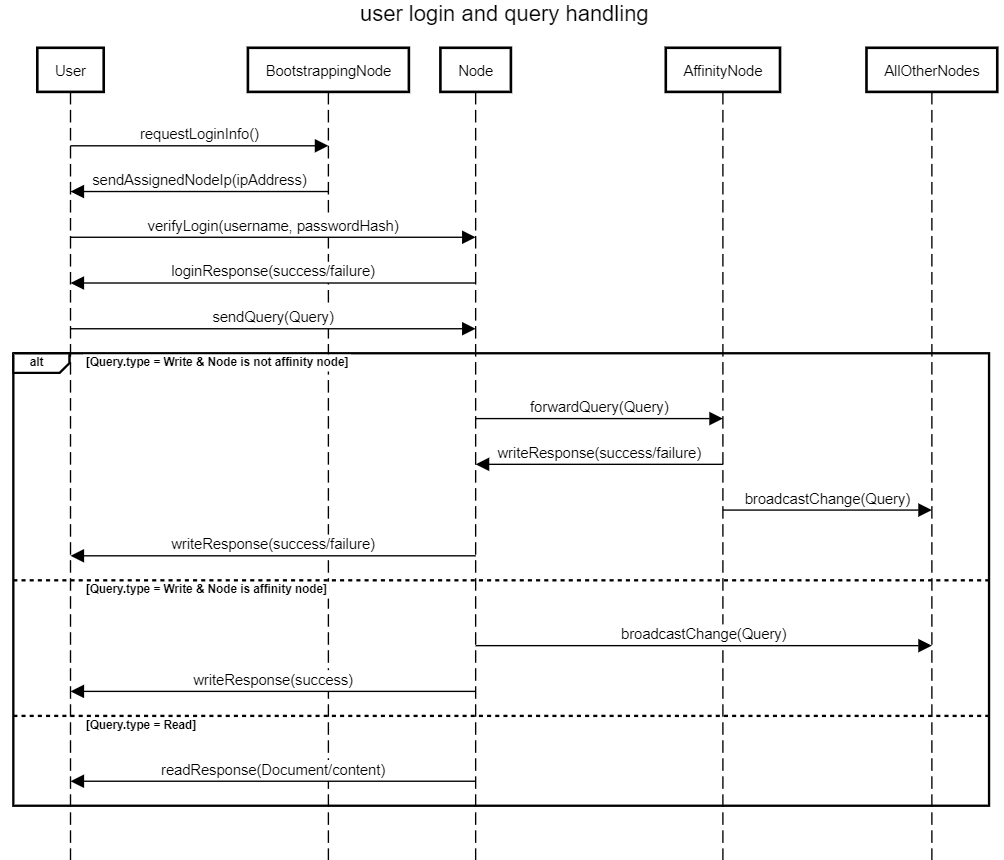
## Overview:



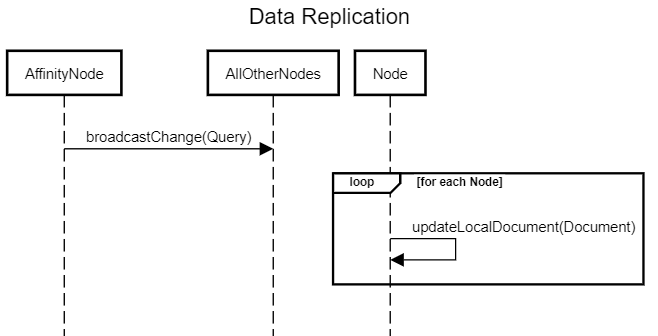
A NoSQL database (DB) is a database that stores data differently from traditional relational DB systems. Thus, NoSQL DBs provide their own APIs for performing queries, instead of using SQL. A cluster-based NoSQL DB is a collection of nodes that can service multiple users, where each node has a replica of the DB. NoSQL DB clusters can have a manager node that works as a central point in the system. We refer to such systems as centralized systems. On the other hand, NoSQL DB systems can be decentralized, where there is no manager node, and instead the nodes in the DB rely on sophisticated schemes for ensuring data consistency and load balance. In this project, I am using Java to build an application that simulates the interaction between users and nodes inside a decentralized NoSQL DB cluster. Below, I will first present the diagrams to make everything clear.

## Diagrams:

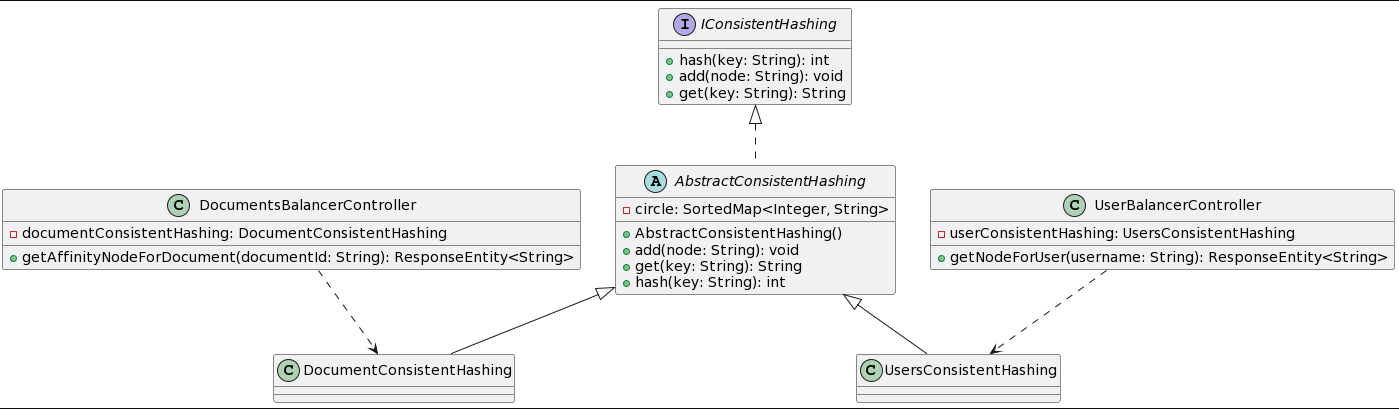
#### Sequence Diagram for user login and query handling:



#### Sequence Diagram for data replication:



#### LOADBALANCE Service:



#### Description:

Consistent Hashing is a technique used in distributing load across a network or infrastructure in a way that the re-distribution of loads, when a node goes down or comes up, is minimized. In the provided LoadBalancer Service Project, Consistent Hashing is employed to achieve effective load balancing across different nodes.

* **Consistent Hashing Implementation**:
  1. A hash function is employed to map both nodes and keys onto a hash circle. This is achieved using the MD5 hashing algorithm.
  2. The hash method in AbstractConsistentHashing class is responsible for generating hash values from input strings (node names and keys).
* **Node Addition:**
  1. Nodes are added to the hash circle with replicas to provide a uniform distribution.
  2. The add method in AbstractConsistentHashing class is used to place each node's replicas onto the hash circle.
* **Key Mapping:**
  1. Keys are mapped to nodes based on their hash values.
  2. The get method in AbstractConsistentHashing class identifies the appropriate node for a given key based on its hash value.
* **Data Structure Used:** A **SortedMap** data structure is utilized to maintain the order of nodes and their replicas based on their hash values in a continuous hash circle. This allows for efficient lookups and minimal re-distribution of keys when nodes are added or removed.

#### Clean Code Principals:

* Meaningful Names: The code employs descriptive names for classes, methods, and variables which make the code self-explanatory.
* Functions: Each function serves a single purpose, such as hashing a key or retrieving a node for a given key.
* Error Handling: Exceptions are caught and meaningful error messages are returned, which is a good practice.

#### Effective Java:

* Error Handling: Exceptions are caught but they could be logged in a better way than just printing the stack trace. Incorporating a logging framework could be beneficial.
* Use of Generics: Generics are defined but not utilized effectively. The AbstractConsistentHashing class has a generic type parameter T that isn't used.

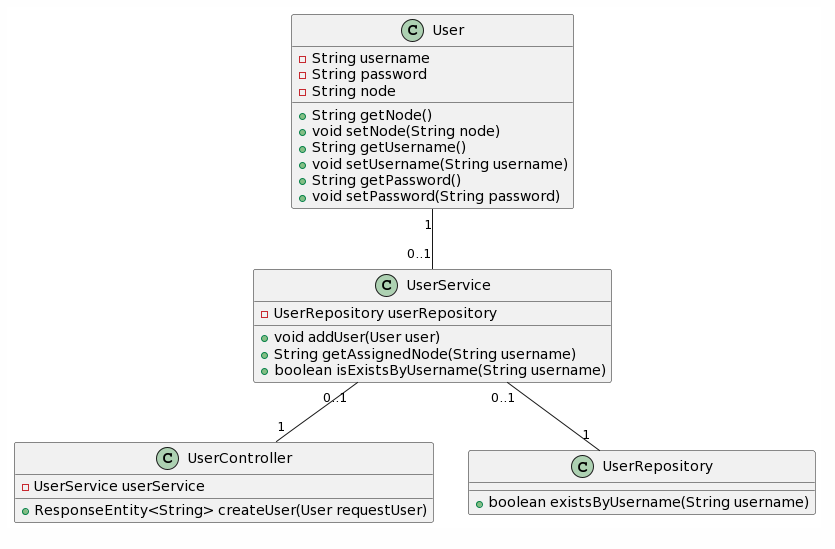
#### SOLID Principals:

* **Open/Closed Principle** (OCP): The code could be made more extensible by allowing the hashing mechanism or the number of replicas to be changed without modifying the existing code.

#### Design Patterns:

* **Factory Pattern**: Could be employed to create instances of AbstractConsistentHashing with different configurations, making the code more flexible and easier to manage

#### LOADBALANCE Service:



**SOLID Principles:**

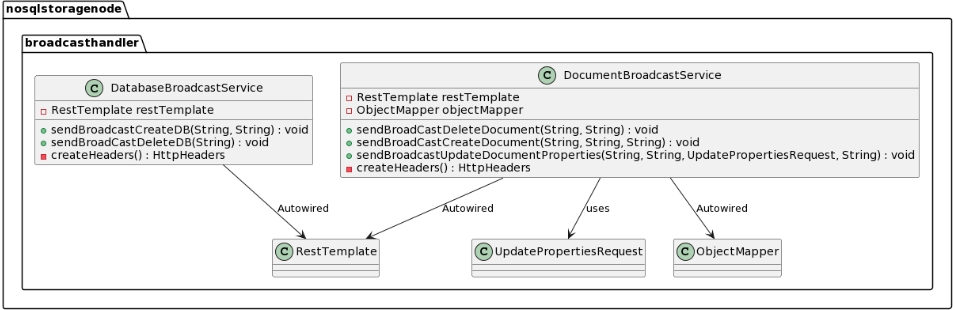
* Single Responsibility Principle (SRP): The classes have a clear responsibility - models hold data, repositories handle data access, services handle business logic, and controllers handle HTTP requests.

**Clean Code Principles:**

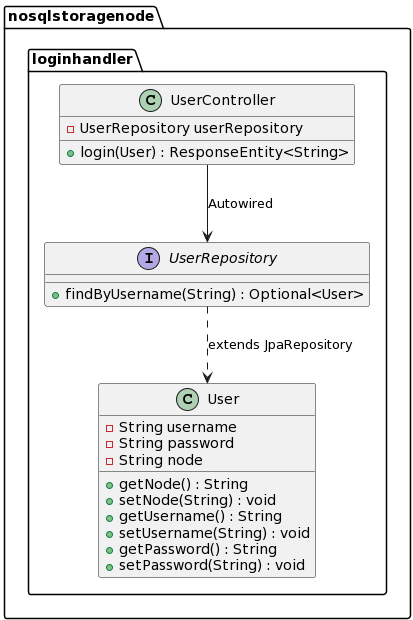
* Meaningful Names: The code employs meaningful names for classes, methods, and variables.
* Small Functions: Methods are broken down into small chunks, each performing a specific task.

#### NoSQLStorageNode Service:

## BroadcastHandler:

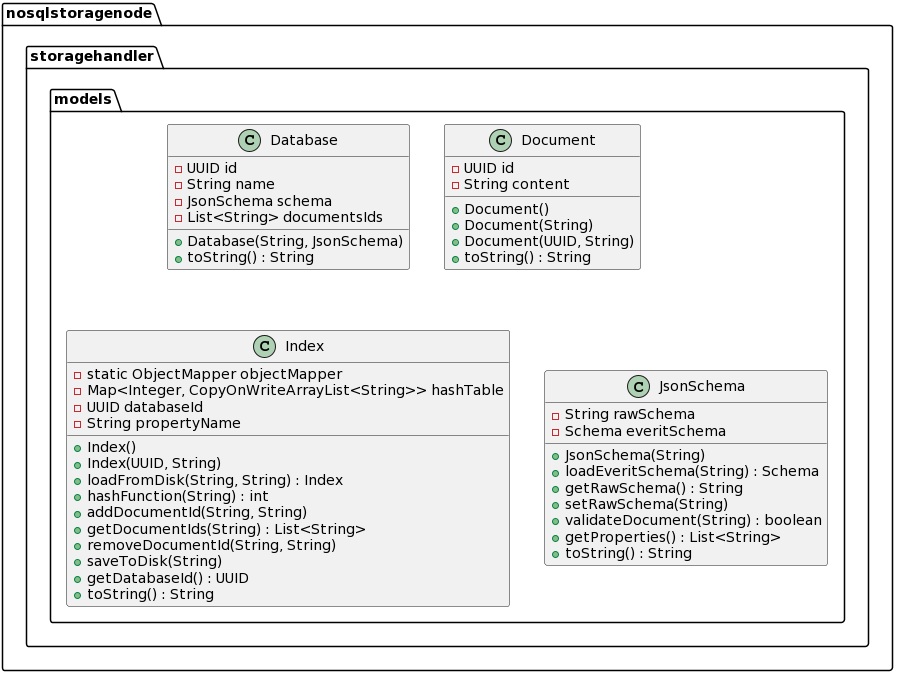


## LoginHandler:



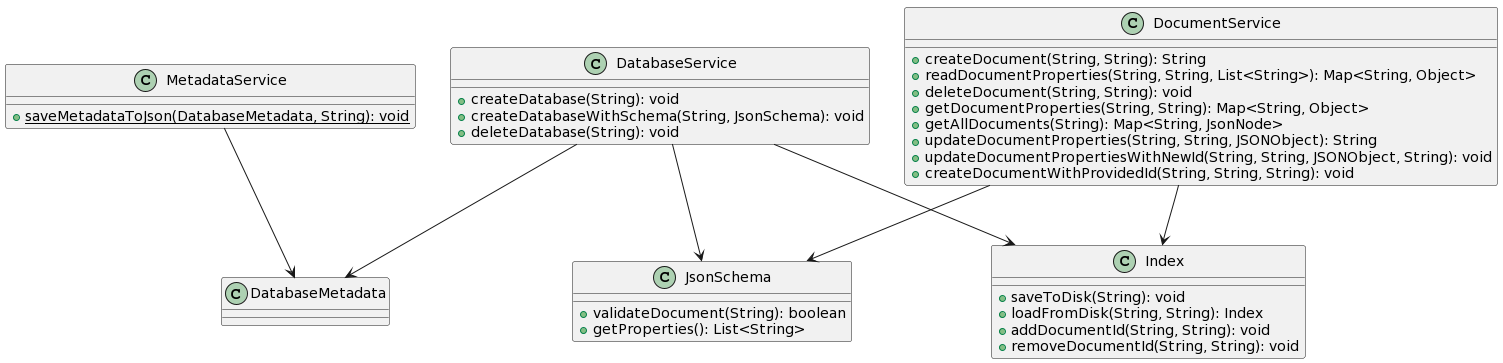
## StorageHandler:

Storage models:



1. Databases
   1. Database\_name
      1. Documents
      2. Schema
      3. Indexes
      4. Metadata

Storage Services:



### Communication Between Nodes:

The primary communication mechanism among nodes is predicated on a RESTful API model. When a client sends a request, it is routed to the affinity node. Post processing, this node issues a broadcast message to other nodes notifying them of the changes.

* **Client Request Handling**:
* Clients interact with nodes through HTTP/HTTPS requests.
* A Load Balancer or a Routing mechanism ensures that the request reaches the affinity node.
* **Processing at the Affinity Node**:
* The affinity node processes the request as per the business logic.
* Any state change as a result of the request is locally committed.
* **Broadcast Messaging**:
* Post local commit, a broadcast message encapsulating the change is sent to all other nodes.

### Test Cases:

* **User Authentication:**
* TC1: Ensure that when a user provides valid credentials, the login request is routed to the assigned node.
* TC2: Verify that an incorrect password results in an authentication failure.
* TC3: Validate that an unrecognized username results in an authentication failure.
* **User Session and Routing:**
* TC5: Upon successful login, ensure that subsequent user requests are routed to the assigned node.
* TC6: If a user is not logged in, ensure that request routing does not occur and a relevant error message is displayed.
* **Document Affinity:**
* TC8: Post login, on querying a document, ensure that the request is processed at the assigned node.
* TC9: Ensure that a non-existent document results in an appropriate error message.
* TC11: Verify that the system can handle simultaneous document queries from multiple users.

### DEVOPS Practices:

* Using Spring and rest controllers
* Clean code
* Using Docker for nodes
* Agile methodology
* Using Git system

### Clean Code Principals:

* **Meaningful Names:**
* Used a descriptive and meaningful name for variables, functions, classes, and modules.
* **Functions:**
* Functions are small and do only one thing.
* Used a descriptive name for functions.
* **Comments:**
* I did not add comments because good code mostly documents itself.
* **Formatting:**
* Follow a consistent coding style.
* Proper indentation, use of whitespace, and organizing code into logical sections are important.
* Keep line lengths manageable
* **Error Handling:**
* Use exceptions rather than return codes for error handling.
* Write try/catch blocks at a level where you can handle them meaningfully.
* Don’t return null, and don’t pass null.
* **Objects and Data Structures:**
* Keep objects small and adhere to the Single Responsibility Principle (SRP).
* Use Object-Oriented Programming (OOP) principles effectively.
* Prefer data encapsulation and data hiding.
* **Code Organization:**
* Organize code in a way that is logical and easy to follow.
* Separate concerns into different modules/classes/functions.
* **Don’t Repeat Yourself (DRY):**
* Reused the code through abstractions.
* **Refactoring:**
* Regularly review and refactor code to improve its structure and readability.

#### SOLID Principals:

* **Single Responsibility Principle (SRP):**
* In my code I tried to make every class or method to have one responsibility.
* **Liskov Substitution Principle (LSP):**
* In my code each derived class can replace its base class without altering the correctness of the program.
* **Interface Segregation Principle (ISP):**
* Tried to not force clients to depend on interfaces they do not use.
* **Dependency Inversion Principle (DIP):**
* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.