Load Balancer

Authors: Kinga Kowal, Sofia Knyshoyid, Dagmara Krenich, Anna Moroń

Load Balancer:

The load balancer is a component responsible for managing access to multiple database nodes.
It acts as an intermediary between the application layer and the physical database instances, providing a single, unified access point for all database operations.
From the application's perspective, the load balancer behaves like a single database.
Internally, it controls how queries are distributed across multiple database nodes, taking into account query type, node availability, and routing strategies.

Responsibilities:

- intercepting all database queries issued by the application
- distinguishing between read operations (SELECT) and write operations (INSERT, UPDATE, DELETE)
- routing read queries to a single selected database node
- propagating write queries to multiple database nodes
- excluding unavailable database nodes from routing decisions
- cooperating with monitoring and recovery components
- providing a stable and transparent API to the application layer

In our project it serves a purpose of middleware between the demo app and multiple database nodes.

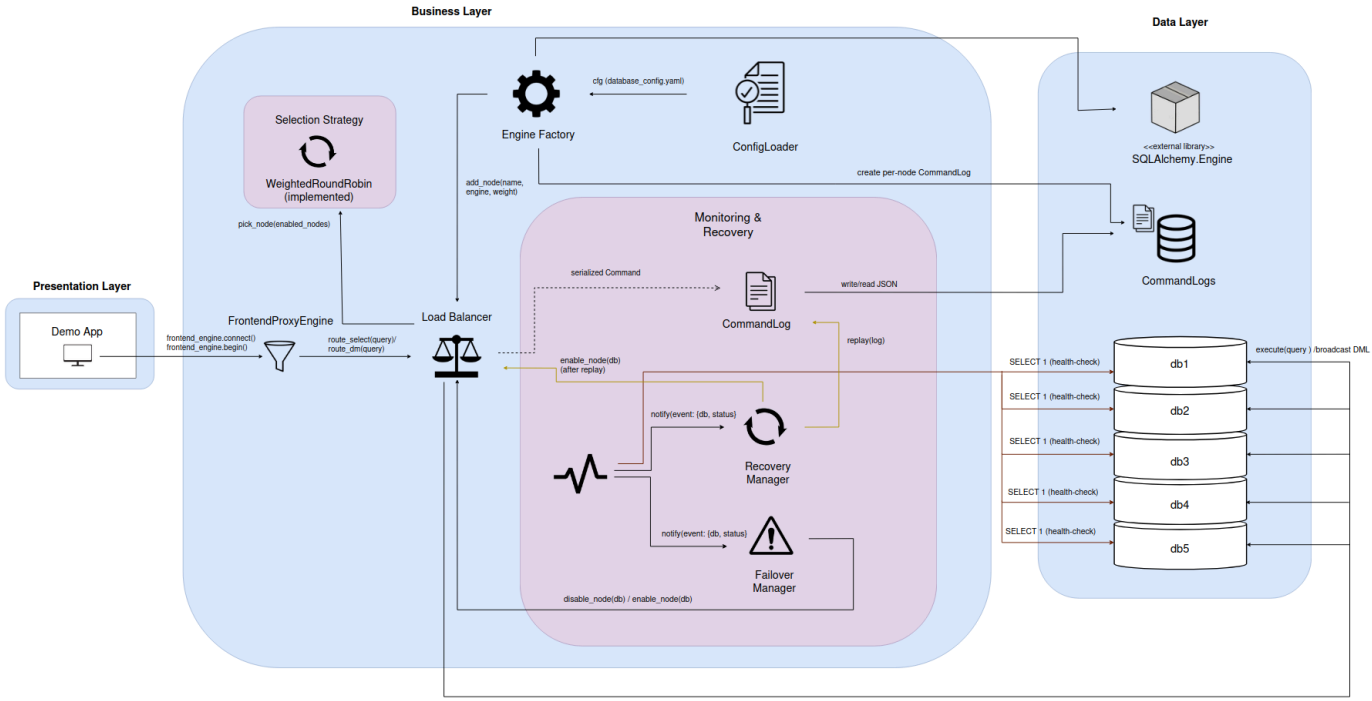
Implementation language: Python

Logical Architecture

The following diagram presents the logical architecture of the load-balancer middleware system. It illustrates how the application is structured into clearly separated layers and how SQL requests flow through the system, from the client application to multiple database nodes.

The architecture is divided into three main layers:

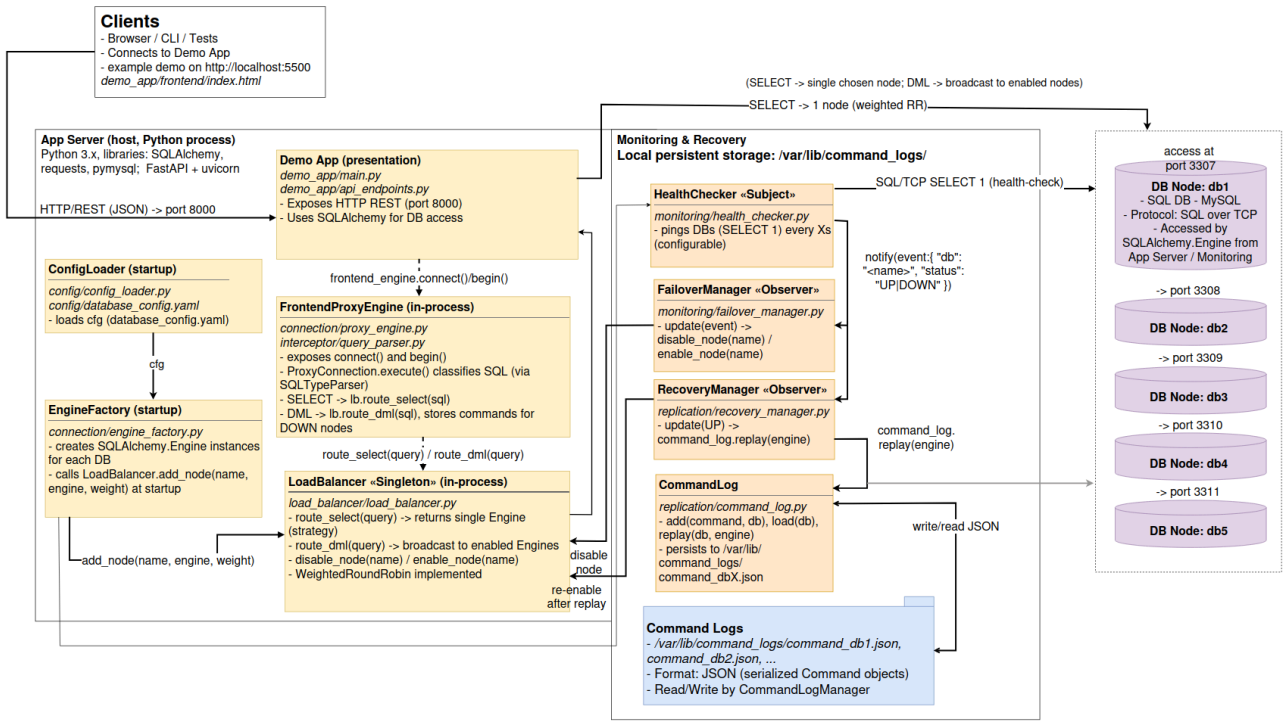
- Presentation Layer, responsible for issuing SQL queries via SQLAlchemy
- Business Layer, which contains the core middleware logic, including SQL interception, load balancing, monitoring, failover, and recovery mechanisms
- Data Layer, which consists of physical database nodes, command logs, and the SQLAlchemy engine abstraction



Physical Architecture

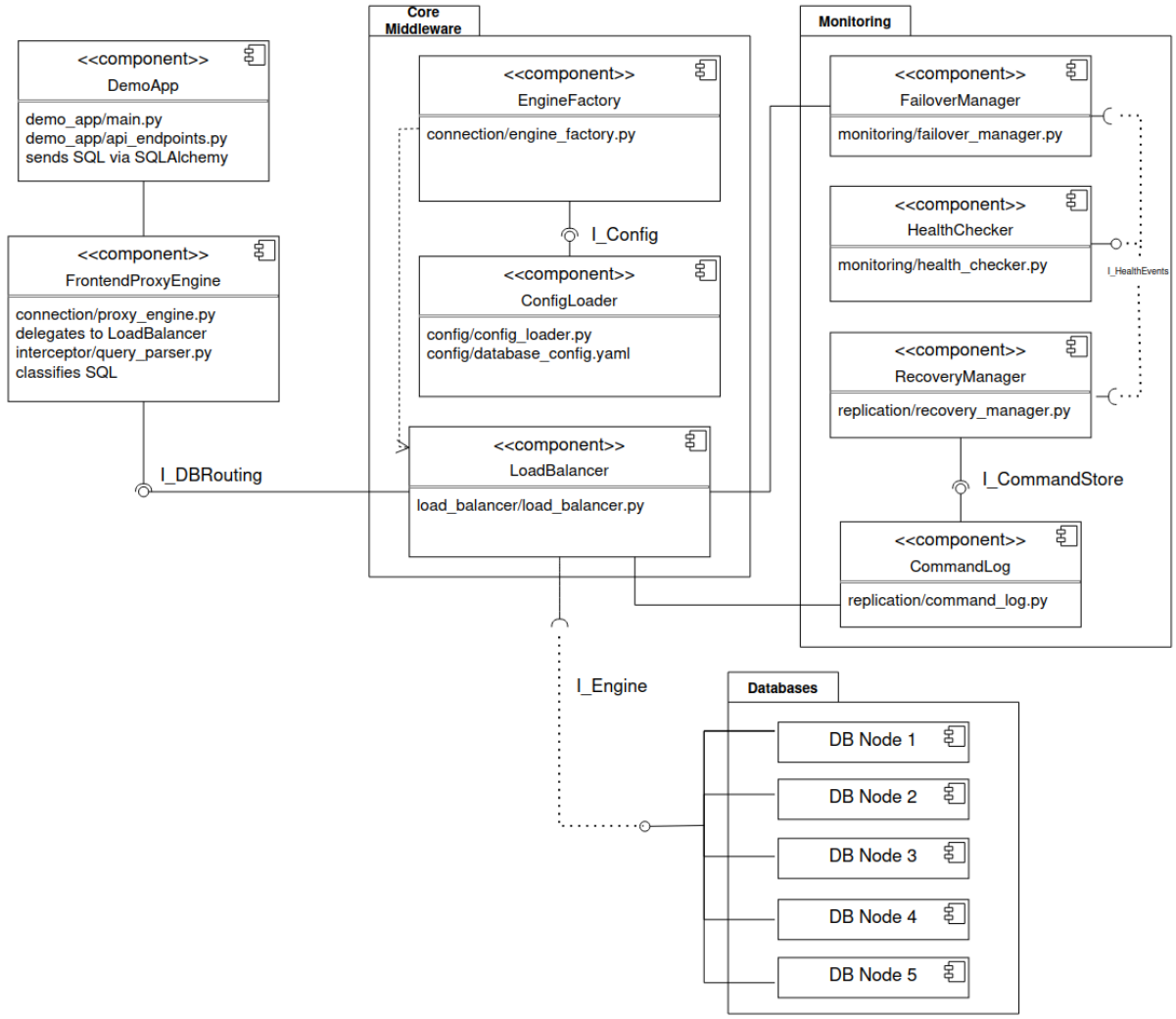
The physical architecture illustrates how the load-balancing middleware, monitoring services, and database nodes are deployed and interact in a real runtime environment. It shows the actual processes, hosts, and data paths involved during execution.
The system is deployed as follows:

- clients can access the system through Demo App's HTTP/REST on port 8000
- The App Server acts as the main runtime environment., it's a singular Python process
- on the same device as the App Server local persistent storage is used to save commang_logs
- the system uses three independent database nodes



Component Diagram

The component diagram shows how the application, SQL interception layer, load balancer, monitoring modules, and database nodes cooperate to provide transparent routing, failover, and recovery in a distributed database system.



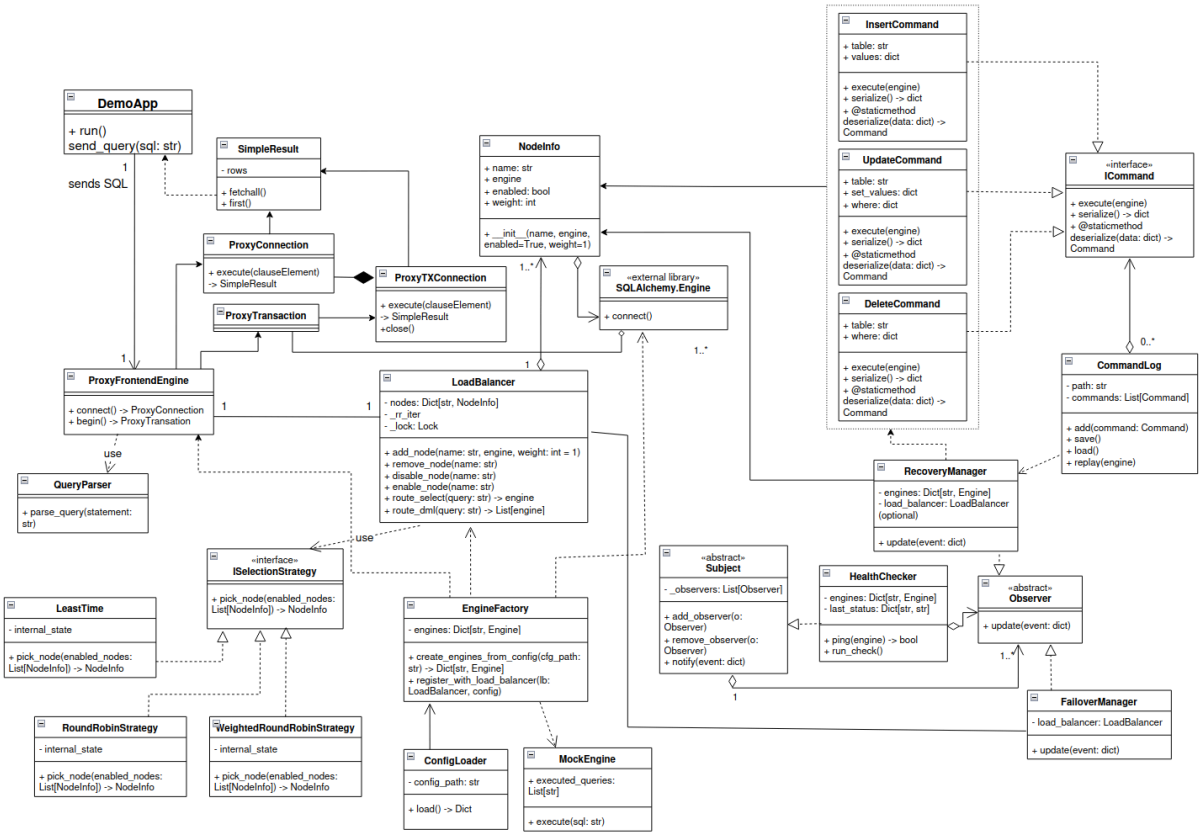
Class diagram

The class diagram presents the main classes that together implement the logic of the load balancer middleware.
It includes components realizing the following design patterns:

- the Observer pattern for monitoring database node availability
- the Command pattern for logging and replaying modifying database operations
- the Strategy pattern with 3 different strategies to choose from - Round Robin, Weighted Round Robin, Least Time

In addition, the diagram contains:

- the LoadBalancer class as the central routing component
- DatabaseNode classes representing individual database instances



Design Patterns

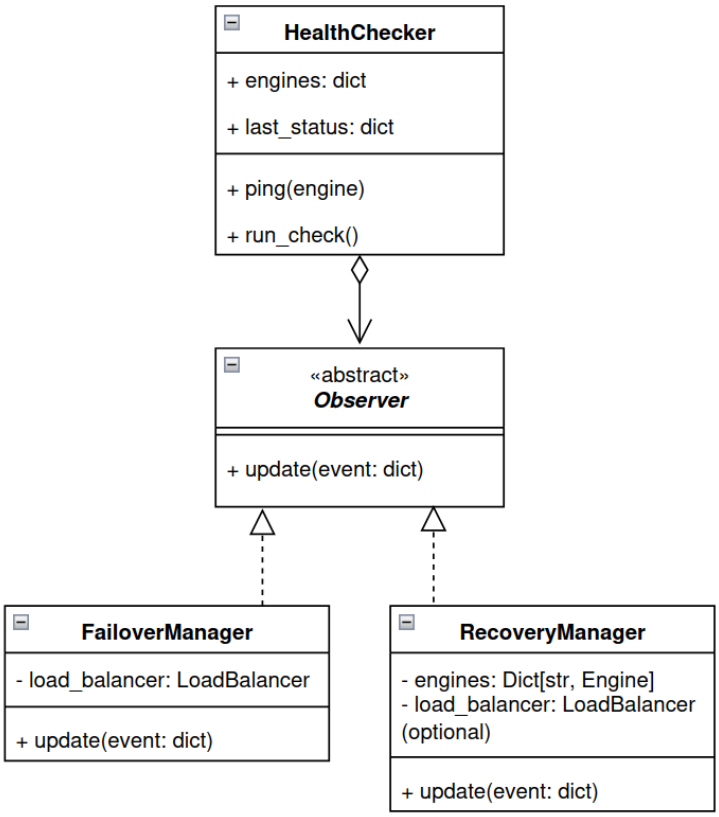
1. Observer

Purpose: Wanting to be able to react to changes in the availability of database nodes. Needing to model one-to-many structure. Separating database monitoring from reactions.

Solution: The pattern allows the HealthChecker (Subject) to notify multiple components (Observers) whenever a database becomes UP or DOWN. This decouples monitoring logic of failover and recovery mechanisms.

Structure:
The Subject is a central component that:
- maintains a list of registered observers
- generates events when something changes
- notifies all observers automatically
In our project, the main Subject is HealthChecker, because it monitors database availability and generates events when state changes.

An Observer is any component that wants to react to database state changes. Observers implement the update(event) method from the abstract *Observer*. Each observer reacts differently:
- FailoverManager - disables a database node when it goes DOWN
- RecoveryManager - starts replaying logs when a database becomes UP



Consequences:
Advantages: Code transparency, stepping closer to single-responsibility design, easier code maintainability.

Costs: Reduced efficiency, an observer notifies all observing objects in a case of a state change (no subscription for topic mechanism).

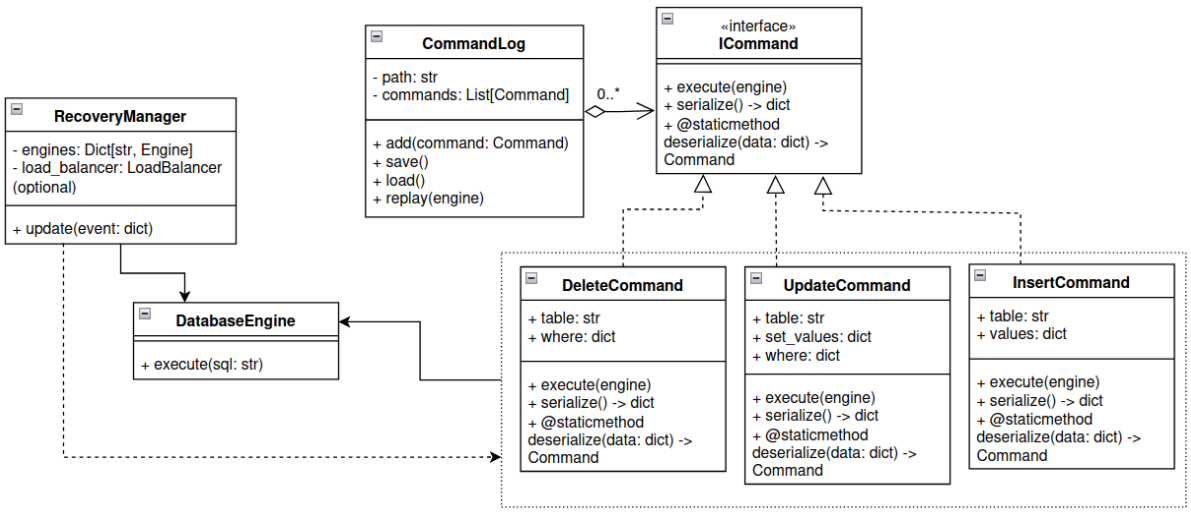
2. Command

Purpose: Turning a database operation (such as INSERT, UPDATE, DELETE) into a standalone object.
If a node becomes temporarily unavailable, modifying database operations can't be performed immediately, but the operation shouldn't be lost.

Solution: Instead of executing SQL immediately, the system wraps each operation inside a command object that can be:
- stored
- serialized
- logged
- transmitted
- replayed later.

This allows our system to persist database modifications even when the database node is DOWN, through replaying all stored commands in order.

Structure:
Command is an abstract class that gets implemented by all specific command classes in our case: by insert, update, delete.
CommandLog serves the purpose of an *Invoker*. It stores a list of commands.
RecoveryManager is our *Client*, as described above, reacts to change of nodes state (up/down) and initiates replay.
DatabaseEngine is our *Receiver* and runs sql only.



Consequences:
Advantages: Code readability, stepping closer to single-responsibility design (decouples request from execution), simple integration to the rest of the system via recoveryManager / Observer, improves fault tolerance and durability.

Costs: Memory and object overhead, command log growth, additional indirection.

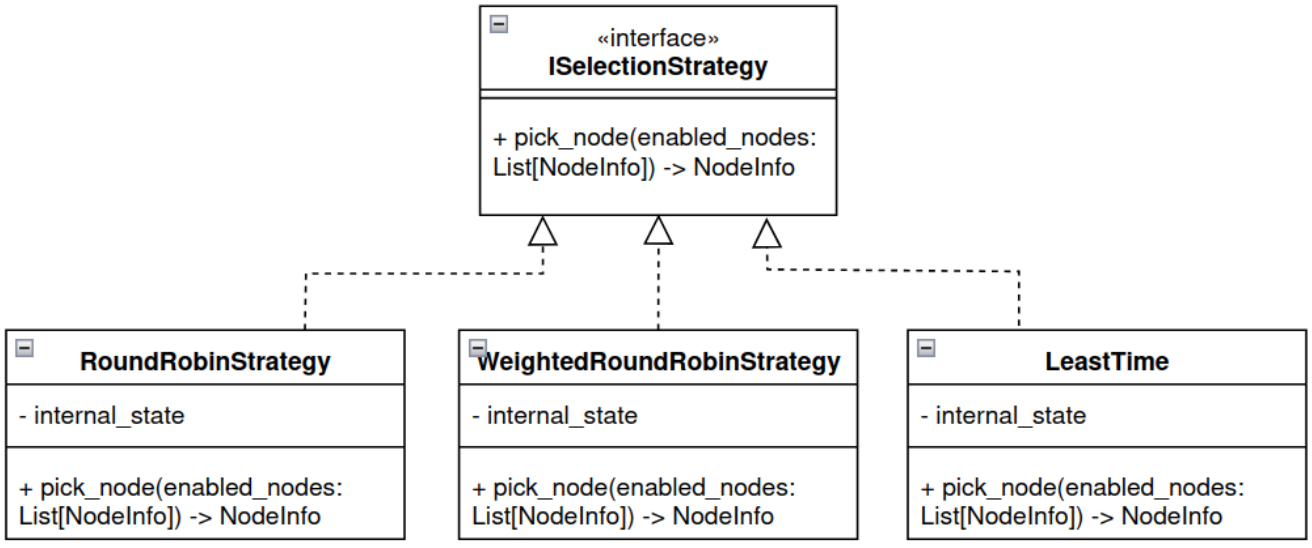
3. Strategy

Purpose: When executing a SELECT command, only one database node is required. We want to have different strategies of choosing the next used database node implemented. The Strategy pattern allows us to define multiple algorithms for selecting the next database node and swap them without modifying the core load balancer logic.

Solution: The Load Balancer depends only on the strategy interface, not on any specific implementation. As a result, strategies can be replaced or extended without changing the load balancer's code.

Structure: Strategy pattern structure includes:

- ISelectionStrategy - defines the abstract method for selecting a database node.
- Concrete Strategies - that implement the selection logic defined in the strategy interface. In our system these are:
    - RoundRobinStrategy - select is executed on available nodes in order,
    - WeightedRoundRobinStrategy - like RoundRobin, but node weights are taken into consideration, nodes with larger weight get picked more often,
    - LeastTime - select is executed on node with the shortest response time.



Consequences:
Advantages: Strategies are easily interchangeable, new strategies can be easily added, it gives configuration flexibility, cleaner code.

Costs: Risk of inconsistent behaviour regarding response time depending on strategy used, potential configuration complexity.