1. Physical Architecture
2. Logical architecture
3. Class diagram
3.1 Total
3.2 Patterns review

1. Physical Architecture

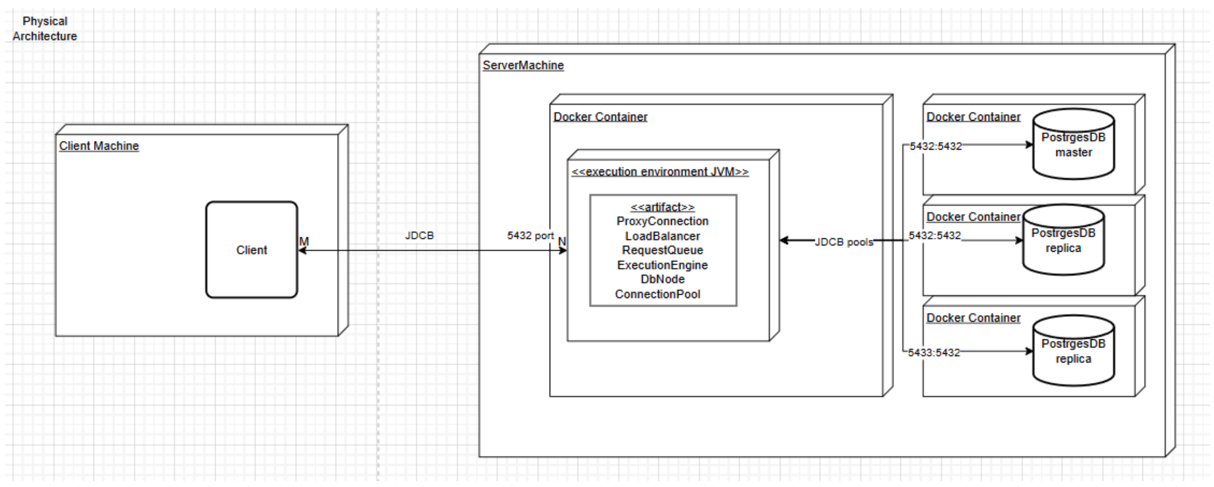
The system is deployed in a client–server architecture. A client application communicates with a server machine over HTTP. The server machine hosts two Docker containers: a Load Balancer container and a PostgreSQL database container.

The Load Balancer container runs a Java-based server application that exposes an HTTP API for database request submission. This container encapsulates the core execution logic, including request scheduling, timeout management, connection pooling, and load-aware dispatching across database nodes.

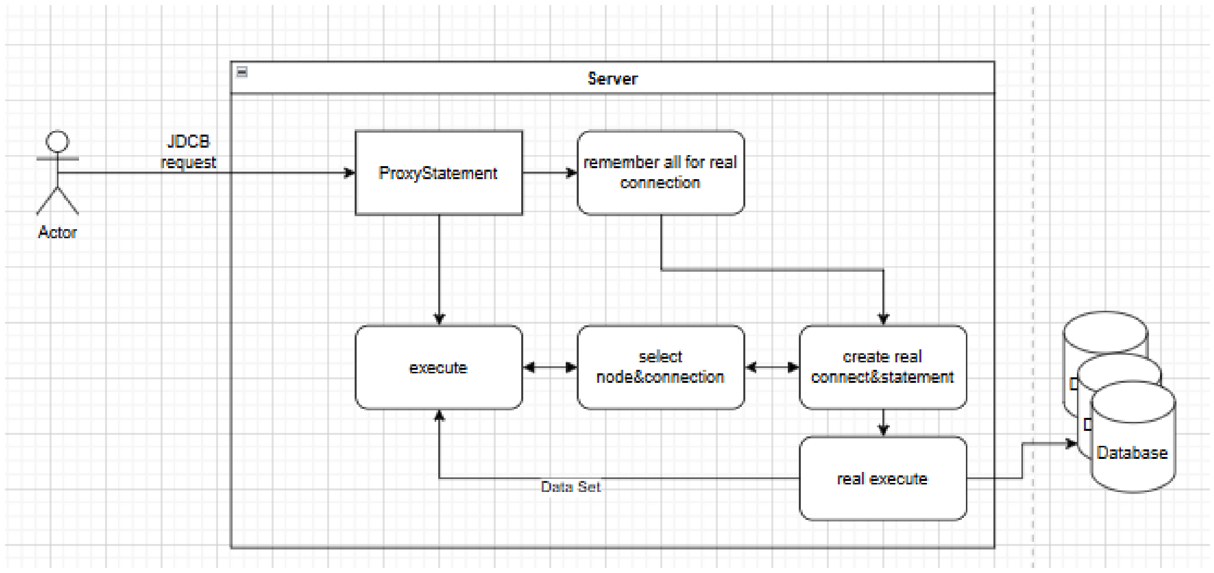
Internally, the server application consists of multiple logical components: a Request Queue, an Execution Engine, a Timeout Manager, and one or more Database Nodes. Each Database Node maintains a pool of JDBC connections to the PostgreSQL container. The database itself runs as an isolated Docker container and is accessed exclusively via JDBC.

Clients submit SQL requests to the server via HTTP. The server enqueues incoming requests and processes them asynchronously. Request execution is performed using pooled database connections and is distributed across available database nodes based on current load. If a request exceeds its execution deadline, it is cancelled and marked as timed out.

All data persistence is handled by PostgreSQL. The server application is stateless with respect to business data and relies on the database container for durability and consistency.



2. Logical architecture



The system follows a layered architecture with a clear separation of responsibilities across four main layers: Presentation, Application, Domain, and Infrastructure.

Presentation Layer

The Presentation Layer is responsible for communication with external clients. Incoming HTTP requests from the client are handled by server-side controllers (or request handlers) that expose the database execution API. This layer performs basic request parsing and validation and delegates all processing to the application layer.

The presentation layer does not contain any business or execution logic. Its sole responsibility is to translate external requests into internal application-level commands and return the corresponding responses or asynchronous results to the client.

Application Layer

The Application Layer implements the system's core use cases related to database request processing. These include submitting a database request, scheduling its execution, dispatching it to an execution engine, handling execution completion, and managing request timeouts.

This layer orchestrates the interaction between the request queue, the execution engine, and the timeout manager. It coordinates asynchronous execution flows, enforces request lifecycle rules, and ensures that state transitions are performed consistently. The application layer also integrates load-balancing decisions by selecting appropriate database nodes based on current load.

Domain Layer

The Domain Layer encapsulates the core execution logic and domain model of the system. It defines the `DbRequest` abstraction, the request state machine (implemented using the State pattern), and the rules governing valid state transitions (e.g. CREATED → QUEUED → ASSIGNED → EXECUTING → COMPLETED / TIMED_OUT).

This layer contains no knowledge of HTTP, threading primitives, or database-specific APIs. It models the request lifecycle, execution semantics, and outcome handling in a technology-agnostic manner. Domain objects expose intent-based operations (such as `markExecuting`, `requeue`, or `timeout`) rather than raw state manipulation.

Infrastructure Layer

The Infrastructure Layer provides concrete technical implementations required by the upper layers. It includes the JDBC-based database connection handling, database node implementations, thread pools and executors for asynchronous execution, and the scheduled timeout mechanism based on `ScheduledExecutorService`.

This layer is also responsible for low-level concurrency primitives, such as atomic state updates, connection pooling, and interaction with the PostgreSQL database. All external system dependencies are isolated in this layer.

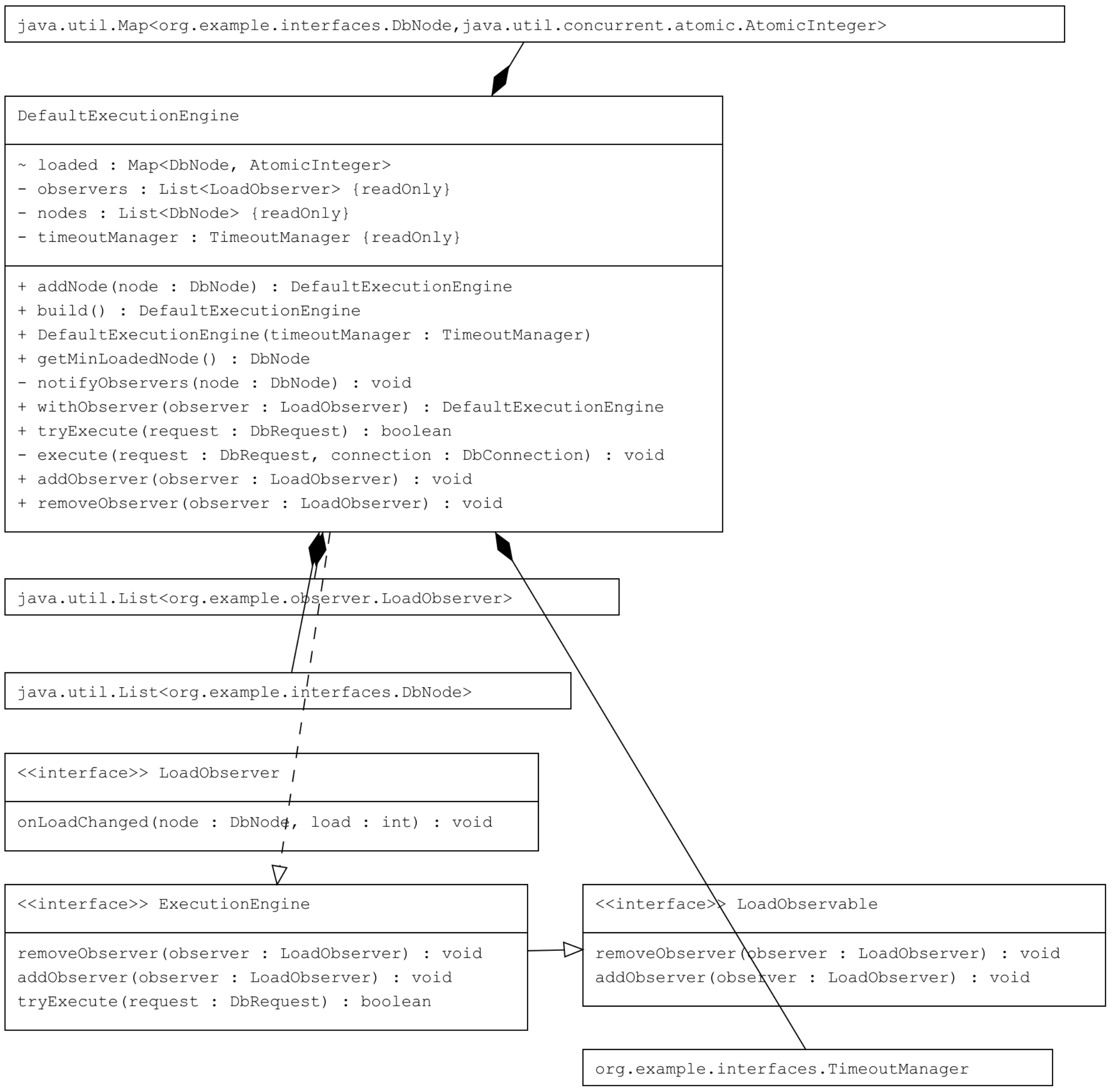
Dependency Rules

Dependencies are strictly directed from outer layers to inner layers. The Presentation Layer depends on the Application Layer, which depends on the Domain Layer. The Infrastructure Layer provides implementations that are used by the Application and Domain layers but does not depend on them.

This structure ensures modularity, testability, and clear isolation of the core execution logic from transport, persistence, and concurrency concerns.

3.Patterns review

Observer



java.util.Map<org.example.interfaces.DbNode,java.util.concurrent.atomic.AtomicInteger>
DefaultExecutionEngine
~ loaded : Map<DbNode, AtomicInteger>
- observers : List<LoadObserver> {readOnly}
- nodes : List<DbNode> {readOnly}
- timeoutManager : TimeoutManager {readOnly}
+ addNode(node : DbNode) : DefaultExecutionEngine
+ build() : DefaultExecutionEngine
+ DefaultExecutionEngine(timeoutManager : TimeoutManager)
+ getMinLoadedNode() : DbNode
- notifyObservers(node : DbNode) : void
+ withObserver(observer : LoadObserver) : DefaultExecutionEngine
+ tryExecute(request : DbRequest) : boolean
- execute(request : DbRequest, connection : DbConnection) : void
+ addObserver(observer : LoadObserver) : void
+ removeObserver(observer : LoadObserver) : void
java.util.List<org.example.observer.LoadObserver>
java.util.List<org.example.interfaces.DbNode>
<<interface>> LoadObserver
onLoadChanged(node : DbNode, load : int) : void
<<interface>> ExecutionEngine
removeObserver(observer : LoadObserver) : void
addObserver(observer : LoadObserver) : void
tryExecute(request : DbRequest) : boolean
<<interface>> LoadObservable
removeObserver(observer : LoadObserver) : void
addObserver(observer : LoadObserver) : void
org.example.interfaces.TimeoutManager

The **Observer** pattern defines a one-to-many dependency between objects so that when one object changes its state, all dependent objects are automatically notified and updated.

In this system, the pattern is used to monitor database node load. The `ExecutionEngine` acts as the *Subject*, while implementations of `LoadObserver` act as *Observers* that are notified whenever the load of a database node changes.

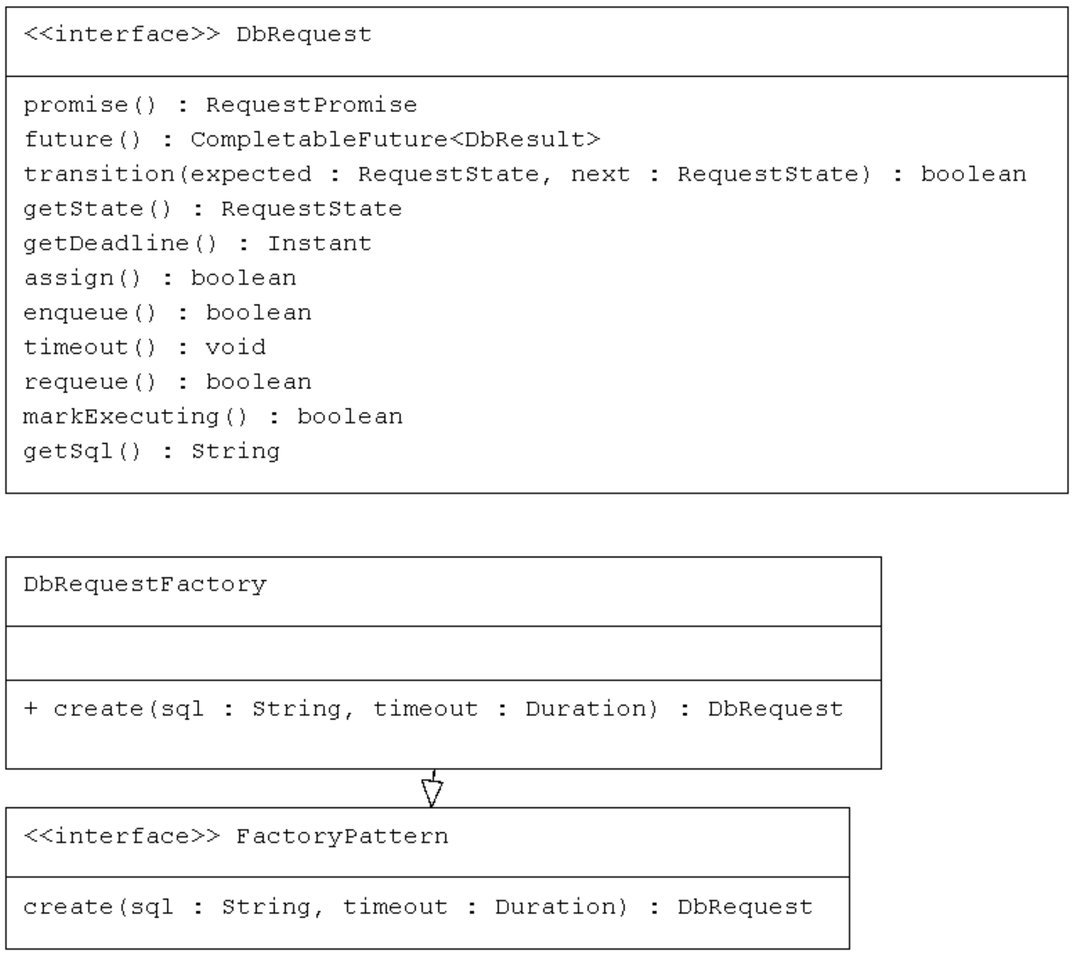
Strengths

- **Loose coupling** between the execution engine and monitoring components
- **High extensibility**: new observers (logging, monitoring, autoscaling) can be added without modifying core logic
- **Reactive behavior**: load changes are propagated immediately
- **Good scalability in functionality**

Potential Issues

- **Debugging complexity** due to indirect control flow
- **Risk of memory leaks** if observers are not properly deregistered
- **Performance degradation** if many observers perform heavy operations
- **No guaranteed notification order**, which may become relevant in advanced scenarios

Factory


<<interface>> DbRequest
promise() : RequestPromise
future() : CompletableFuture<DbResult>
transition(expected : RequestState, next : RequestState) : boolean
getState() : RequestState
getDeadline() : Instant
assign() : boolean
enqueue() : boolean
timeout() : void
requeue() : boolean
markExecuting() : boolean
getSql() : String
DbRequestFactory
+ create(sql : String, timeout : Duration) : DbRequest
<<interface>> FactoryPattern
create(sql : String, timeout : Duration) : DbRequest

The **Factory** pattern encapsulates object creation logic, allowing clients to remain independent of the concrete classes being instantiated.

In the system, factories are used to create:

- `DbRequest` instances (`DbRequestFactory`)
- Database connections (`DbConnectionFactory`)

This design allows the system to change or extend creation logic (e.g., JDBC to another driver) without impacting the rest of the codebase.

Strengths

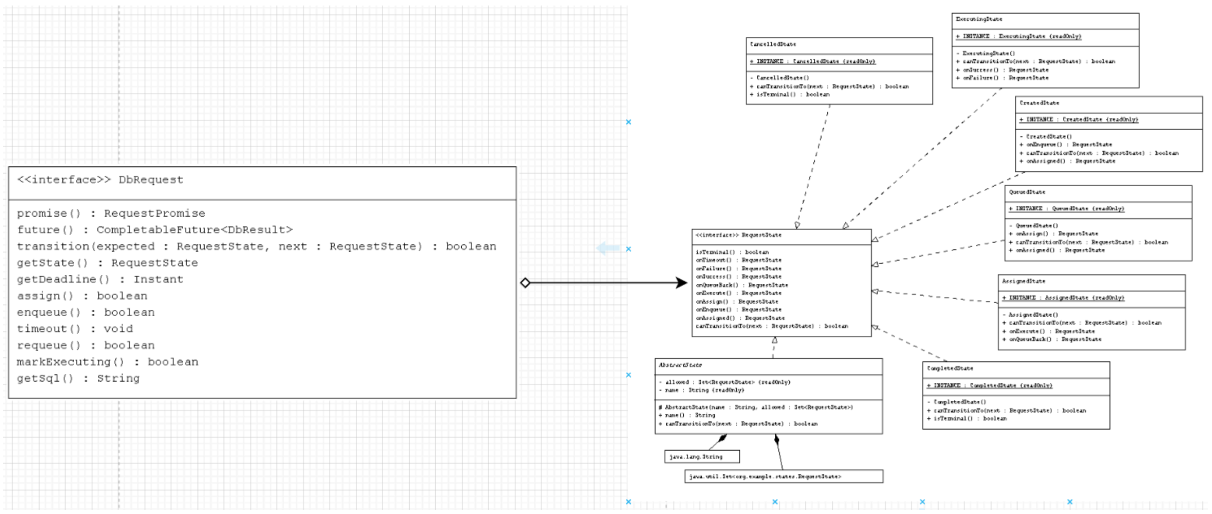
- **Encapsulation of creation logic**
- **Ease of replacement or extension** of implementations

- **Improved testability** through dependency substitution
- **Compliance with the Open/Closed Principle**

Potential Issues

- **Increased number of classes** in large systems
- **Hidden complexity** if factory logic grows too complex
- **Over-abstraction** when factories create only a single concrete type
- Risk of becoming an **unnecessary indirection** if misused

State Pattern



<<interface>> DbRequest
promise() : RequestPromise
future() : CompletableFuture<DbResult>
transition(expected : RequestState, next : RequestState) : boolean
getState() : RequestState
getDeadline() : Instant
assign() : boolean
enqueue() : boolean
timeout() : void
requeue() : boolean
markExecuting() : boolean
getSql() : String
<<interface>> RequestState
isTerminal() : boolean
onTimeout() : RequestState
onFailure() : RequestState
onSuccess() : RequestState
onQueueBack() : RequestState
onExecute() : RequestState
onAssign() : RequestState
onEnqueue() : RequestState
onAssigned() : RequestState
canTransitionTo(next : RequestState) : boolean
CancelledState
+ INSTANCE : CancelledState {readOnly}
- CancelledState()
+ canTransitionTo(next : RequestState) : boolean
+ isTerminal() : boolean
ExecutingState
+ INSTANCE : ExecutingState {readOnly}
- ExecutingState()
+ canTransitionTo(next : RequestState) : boolean
+ onSuccess() : RequestState
+ onFailure() : RequestState
CreatedState
+ INSTANCE : CreatedState {readOnly}
- CreatedState()
+ onEnqueue() : RequestState
+ canTransitionTo(next : RequestState) : boolean
+ onAssigned() : RequestState
QueuedState
+ INSTANCE : QueuedState {readOnly}
- QueuedState()
+ onAssign() : RequestState
+ canTransitionTo(next : RequestState) : boolean
+ onAssigned() : RequestState
AssignedState
+ INSTANCE : AssignedState {readOnly}
- AssignedState()
+ canTransitionTo(next : RequestState) : boolean
+ onExecute() : RequestState
+ onQueueBack() : RequestState
AbstractState
- allowed : Set<RequestState> {readOnly}
- name : String {readOnly}
# AbstractState(name : String, allowed : Set<RequestState>)
+ name() : String
+ canTransitionTo(next : RequestState) : boolean
CompletedState
+ INSTANCE : CompletedState {readOnly}
- CompletedState()
+ canTransitionTo(next : RequestState) : boolean
+ isTerminal() : boolean
java.lang.String
java.util.Set<org.example.states.RequestState>

The **State** pattern allows an object to alter its behavior when its internal state changes. From the outside, the object appears to change its class.

In this system, the pattern is used to model the lifecycle of a `DbRequest`. Each state (`Created`, `Queued`, `Executing`, `Completed`, `TimedOut`, etc.) encapsulates valid transitions and state-specific behavior, replacing conditional logic and `enum`-based state handling.

Strengths

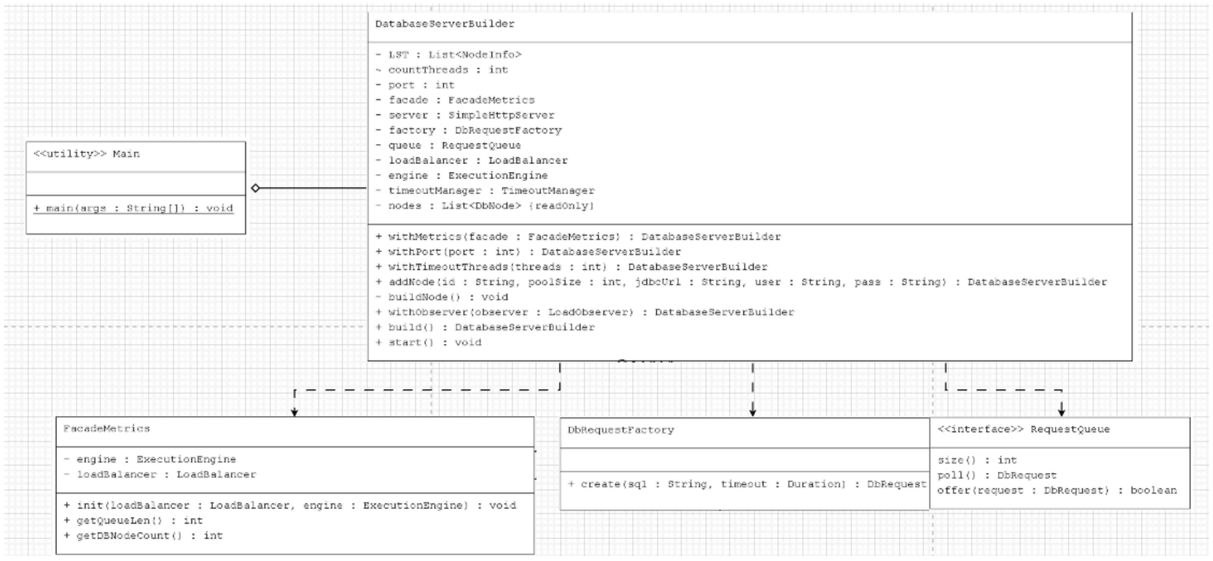
- **Explicit and well-defined lifecycle model**
- **Prevention of invalid state transitions**
- **Reduction of complex conditional logic**

- **Improved readability and maintainability**
- **Adherence to the Single Responsibility Principle**

Potential Issues

- **Increased number of classes**
- **Higher entry barrier** for new developers
- **Overengineering** for simple state machines
- Risk of **state explosion** if not carefully designed

Builder


DatabaseServerBuilder
- LST : List<NodeInfo>
~ countThreads : int
- port : int
- facade : FacadeMetrics
- server : SimpleHttpServer
- factory : DbRequestFactory
- queue : RequestQueue
- loadBalancer : LoadBalancer
- engine : ExecutionEngine
- timeoutManager : TimeoutManager
- nodes : List<DbNode> {readOnly}
+ withMetrics(facade : FacadeMetrics) : DatabaseServerBuilder
+ withPort(port : int) : DatabaseServerBuilder
+ withTimeoutThreads(threads : int) : DatabaseServerBuilder
+ addNode(id : String, poolSize : int, jdbcUrl : String, user : String, pass : String) : DatabaseServerBuilder
- buildNode() : void
+ withObserver(observer : LoadObserver) : DatabaseServerBuilder
+ build() : DatabaseServerBuilder
+ start() : void
<<utility>> Main
+ main(args : String[]) : void
FacadeMetrics
- engine : ExecutionEngine
- loadBalancer : LoadBalancer
+ init(loadBalancer : LoadBalancer, engine : ExecutionEngine) : void
+ getQueueLen() : int
+ getDBNodeCount() : int
DbRequestFactory
+ create(sql : String, timeout : Duration) : DbRequest
<<interface>> RequestQueue
size() : int
poll() : DbRequest
offer(request : DbRequest) : boolean

The **Builder** pattern is used to construct complex objects step by step, allowing fine-grained configuration before the object becomes operational.

In this system, the pattern is applied during the initialization of the `ExecutionEngine`, where multiple components (database nodes, observers, timeout handling) must be configured before the engine is built and started.

Strengths

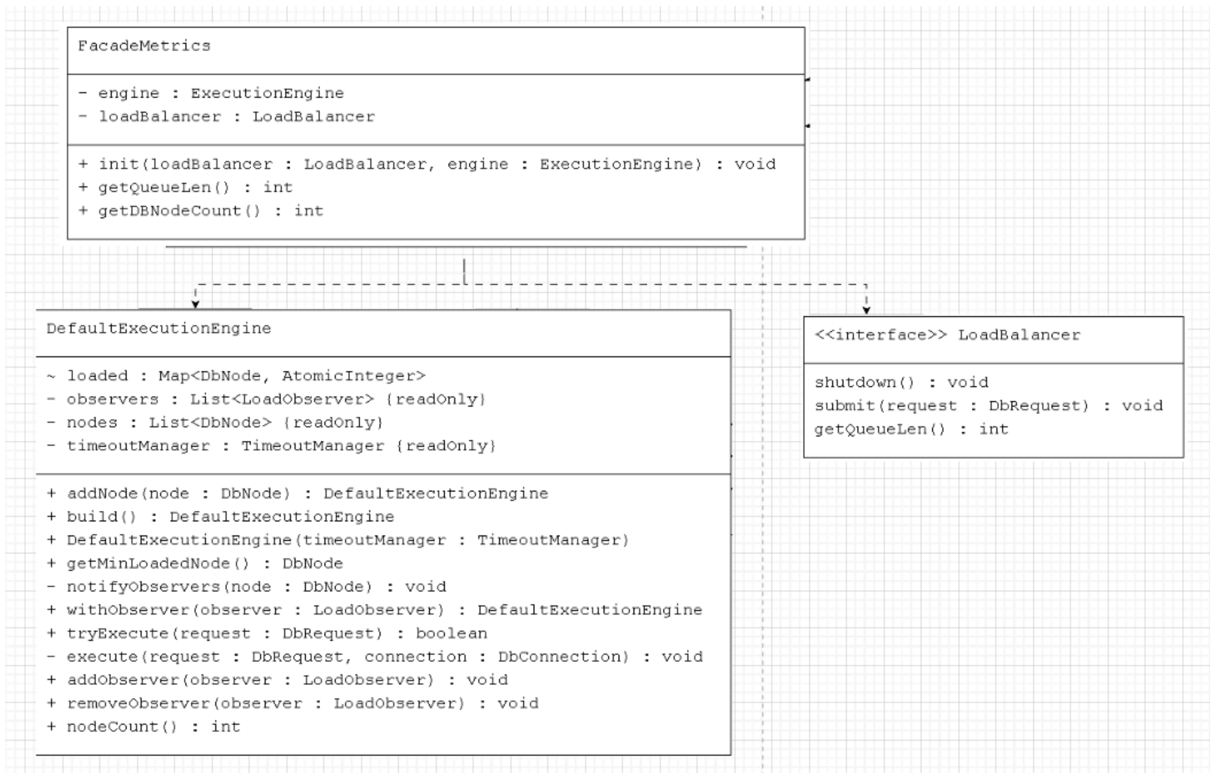
- **Clear and readable object construction**
- **Avoidance of telescoping constructors**

- **Flexible configuration process**
- **Supports immutable objects after construction**

Potential Issues

- **Additional implementation overhead**
- **Risk of partially configured objects** without proper validation
- **Unnecessary complexity** for simple objects
- May obscure **mandatory dependencies** if the builder is too permissive

Facade


FacadeMetrics
- engine : ExecutionEngine
- loadBalancer : LoadBalancer
+ init(loadBalancer : LoadBalancer, engine : ExecutionEngine) : void
+ getQueueLen() : int
+ getDBNodeCount() : int
DefaultExecutionEngine
~ loaded : Map<DbNode, AtomicInteger>
- observers : List<LoadObserver> {readOnly}
- nodes : List<DbNode> {readOnly}
- timeoutManager : TimeoutManager {readOnly}
+ addNode(node : DbNode) : DefaultExecutionEngine
+ build() : DefaultExecutionEngine
+ DefaultExecutionEngine(timeoutManager : TimeoutManager)
+ getMinLoadedNode() : DbNode
- notifyObservers(node : DbNode) : void
+ withObserver(observer : LoadObserver) : DefaultExecutionEngine
+ tryExecute(request : DbRequest) : boolean
- execute(request : DbRequest, connection : DbConnection) : void
+ addObserver(observer : LoadObserver) : void
+ removeObserver(observer : LoadObserver) : void
+ nodeCount() : int
<<interface>> LoadBalancer
shutdown() : void
submit(request : DbRequest) : void
getQueueLen() : int

The **Facade pattern** provides a simplified interface to a complex subsystem, allowing clients to interact with the system without needing to understand its internal complexity. In this system, the pattern is applied to the interaction with the database and request handling subsystems, where multiple classes (such as `DbRequest`, `RequestPromise`, and `DbRequestImplementation`) work together to fulfill client operations. The facade exposes a clean, high-level API, hiding the details of object creation, configuration, and coordination.

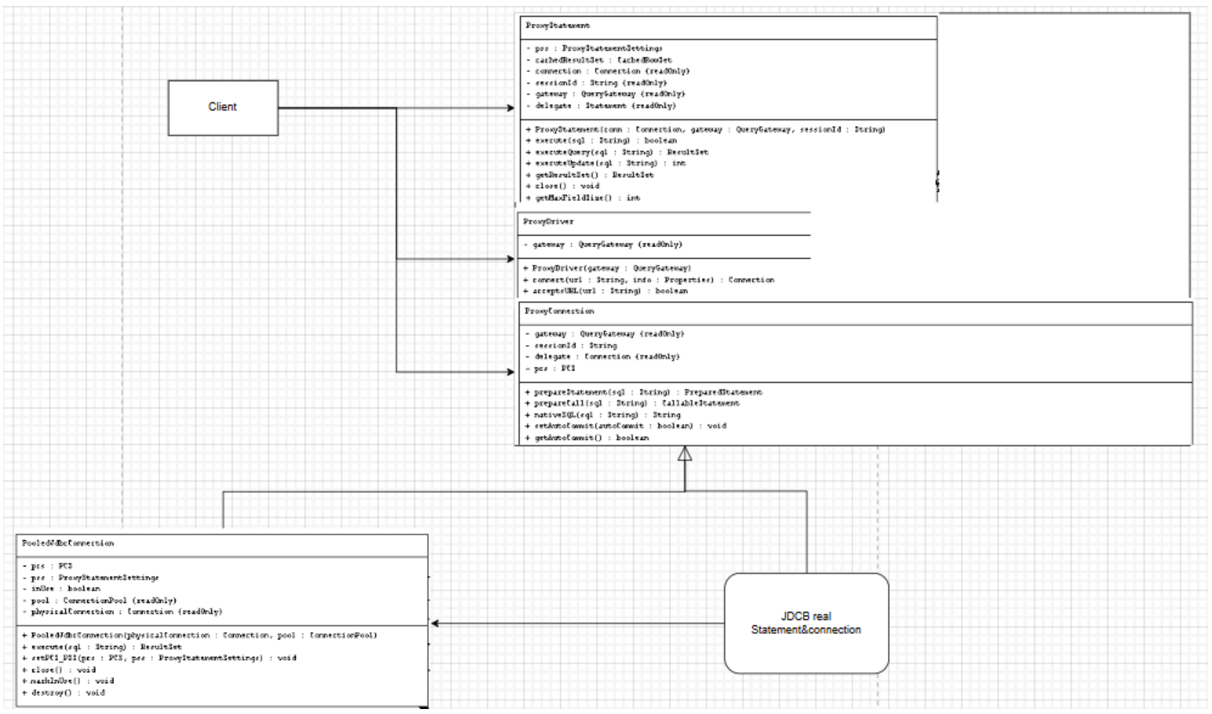
Strengths

- Simplifies client interaction with complex subsystems
- Reduces coupling between clients and subsystem classes
- Improves readability and maintainability of code
- Encourages a clear separation between interface and implementation

Potential Issues

- Can become a “god object” if it grows too large
- May hide important functionality or flexibility of the underlying subsystem
- Adds an additional layer of abstraction, which can be unnecessary for very simple systems
- Risk of oversimplifying subsystem behavior, leading to incomplete usage in advanced scenarios

Proxy Pattern


Client
ProxyStatement
- pss : ProxyStatementSettings
- cachedResultSet : CachedRowSet
- connection : Connection (readOnly)
- sessionId : String (readOnly)
- gateway : QueryGateway (readOnly)
- delegate : Statement (readOnly)
+ ProxyStatement(conn : Connection, gateway : QueryGateway, sessionId : String)
+ execute(sql : String) : boolean
+ executeQuery(sql : String) : ResultSet
+ executeUpdate(sql : String) : int
+ getResultSet() : ResultSet
+ close() : void
+ getMaxFieldSize() : int
ProxyDriver
- gateway : QueryGateway (readOnly)
+ ProxyDriver(gateway : QueryGateway)
+ connect(url : String, info : Properties) : Connection
+ acceptsURL(url : String) : boolean
ProxyConnection
- gateway : QueryGateway (readOnly)
- sessionId : String
- delegate : Connection (readOnly)
- pcs : PCS
+ prepareStatement(sql : String) : PreparedStatement
+ prepareCall(sql : String) : CallableStatement
+ nativeSQL(sql : String) : String
+ setAutoCommit(autoCommit : boolean) : void
+ getAutoCommit() : boolean
PooledJdbcConnection
- pcs : PCS
- pss : ProxyStatementSettings
- inUse : boolean
- pool : ConnectionPool (readOnly)
- physicalConnection : Connection (readOnly)
+ PooledJdbcConnection(physicalConnection : Connection, pool : ConnectionPool)
+ execute(sql : String) : ResultSet
+ setPCS_PSS(pcs : PCS, pss : ProxyStatementSettings) : void
+ close() : void
+ markInUse() : void
+ destroy() : void
JDCB real Statement&connection

The Proxy pattern provides a surrogate or placeholder object that controls access to another object.
In this system, the pattern is applied through `ProxyDriver`, `ProxyConnection`, and

potentially `ProxyStatement`, which wrap the real PostgreSQL JDBC driver and connections.

Instead of allowing the client to communicate directly with the PostgreSQL driver, the proxy intercepts database calls and adds additional behavior such as:

- load balancing
- routing (master vs replica)
- metrics collection
- request monitoring
- fault handling

The proxy exposes the same `java.sql.Connection` interface, so clients remain unaware of the underlying routing and control logic.

Potential Issues

Added complexity

The proxy layer introduces additional classes and control flow, making debugging more complex.

Hidden performance cost

Extra routing logic and object wrapping may introduce slight overhead.

Risk of tight coupling inside proxy

If too much logic is added (routing + metrics + retries + caching), the proxy can become overloaded.

Harder error tracing

Exceptions may originate from underlying connections but appear through proxy layers, complicating stack traces.

Risk of incorrect abstraction

If proxy logic does not perfectly mirror JDBC behavior, subtle bugs may occur.

Strengths

Transparent behavior extension

Clients interact with the standard JDBC API (`Connection`, `Statement`) without knowing a proxy is involved.

Centralized control of database access

All queries pass through a single control point, enabling routing and monitoring.

Supports cross-cutting concerns

Adds features like:

- logging
- metrics
- load balancing
- failover handling

without modifying client code.

Enables dynamic behavior

The proxy can:

- route reads to replicas
- route writes to master
- switch nodes dynamically

Improves scalability

By abstracting node selection, the system can scale horizontally without changing application code.