Distributed Systems and Middleware Technologies Project Report

Erldb: A blazingly fast distributed database

Lorenzo Catoni Leonardo Giovannoni Alessandro Versari

Contents

1 Introduction

This system is a **distributed SQLite database** engineered for *redundancy* and *fault tolerance*. The core functionality is to allow for robust data storage and retrieval across a **distributed network**, ensuring **high availability** and **strict consistency**. The system supports all standard database operations such as **CREATE, READ, UPDATE, and DELETE (CRUD)**, although with certain limitations.

The system uses the **Raft consensus algorithm**, implemented through the *Ra Erlang library*, to manage concurrency. This ensures **total consistency** across the distributed database by committing changes only after a *majority consensus* is reached. The cluster maintains operation as long as more than half the nodes are connected, allowing the system to *tolerate faults effectively*, for these reasons this application is characterized as **CP (Consistency and Partition Tolerance)** in the CAP theorem.

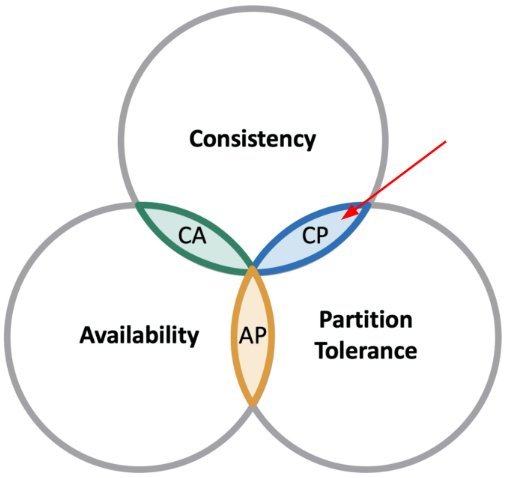


Figure 1: Our system is CP

In order to interact with a SQLite database from Erlang we implemented a library written in **Rust** that provides a *native interface* to the *rusqlite*[1] library. The *Rustler*[2] library was used to make the functions available to Erlang.

We implemented a **JDBC driver in Kotlin** that acts as a *proxy* to the Erlang nodes, ensuring that interactions with the distributed DBMS are transparent to developers working with *Java or Kotlin*, so much so that the driver can be used with any application that uses JDBC, like the *Hibernate ORM*.

We also implemented a **web server in Kotlin** with the *Spring Boot framework* that provides a *REST API* to interact with the database and add *basic authentication* to the system.

Finally the system will be accessible through a **web-based client application**. This application enables *normal users* to have access to specific pre-set database files for controlled data interaction. In contrast, *admin users* possess the authority to create and delete databases and also dictate the database access permission for different users, ensuring a **high level of customization and control** over the system's data management.

[1]An ergonomic wrapper for using SQLite from Rust: `https://docs.rs/rusqlite/latest/rusqlite/`
[2]A library for writing Erlang NIFs in safe Rust code: `https://github.com/rusterlium/rustler`

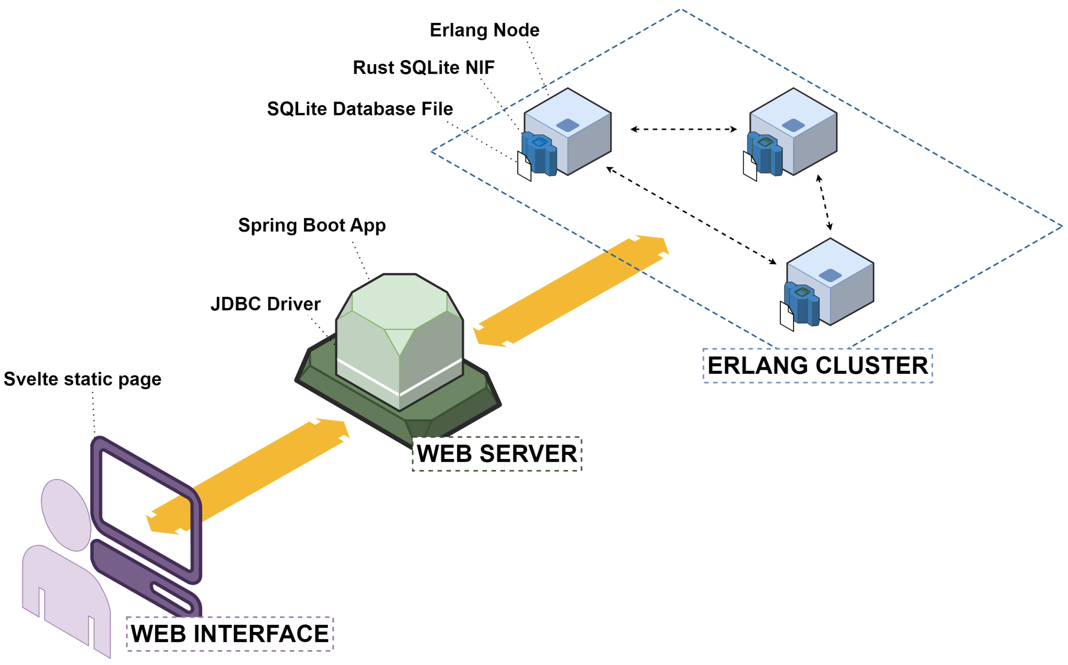


Figure 2: System overview

2 Native bindings

We decided to implement bindings to native code for sqlite, in order to achieve more flexibility over the available third-party libraries. We based our native code implementation on the popular Rust bindings library **rusqlite**, which is in active development with over 100 contributors on GitHub, and has almost all functionalities that C SQLite APIs provide. This allows us to write the code in *safe Rust*, ensuring that many common C bugs are avoided, and we don't have to deal with dangling pointers (however, in case of real need, we could call SQLite's C API, leveraging Rust FFI through rusqlite).

Due to SQLite's nature of single process (and thread) APIs, we adopted a blend of the *Command Pattern* (where each operation is encapsulated as an object or message) and the *Actor Model* (where entities communicate via messages). This approach is similar to what is done by another popular library for Rust, **sqlx**, which provides bindings to SQLite in a more abstracted way than rusqlite, and is used for asynchronous programming purposes.

The basic idea is to let Erlang code create a context, which corresponds to a thread under the hood, and create a channel for each connection or statement. We could achieve a similar result using a single thread and other green threads, using in our case, as for other parts of the code, the *tokio* library for Rust. However, the problem with this approach is that we need a thread-safe version of connections and statements. SQLite's connections and statements are intended for a single thread scenario (in Rust terms, connections and statements which are respectively *!Sync* (can't be shared among threads) and *!Send* (ownership can't be moved among threads) + *!Sync*, so can't be sent or shared by threads safely).

To manage concurrency inside the worker thread, we used the *tokio* library, which allows us to create many tasks, each executing on the worker thread. We have for each statement and for each connection a spawned task, so that many tasks, such as a statement, can be stopped and then resumed when needed without incurring the overhead of having a dedicated thread for different flows of execution.

The use of Rust allowed us to obtain a multithreaded (two threads, actually) library with compile-time guarantees of data race-free code. Each object provided to Erlang is reference-counted, so when the object is no longer reachable, the garbage collector will finalize it, ensuring that the *drop* function will be called, and there won't be any memory/resource leak, eliminating the need for manual cleanup.

Another reason to create native bindings is to define with better granularity which operations are consistent in a multithreaded environment. Erlang processes are, after all, distributed on Unix threads, so we need to guarantee thread safety when creating native bindings. We used the *rustler* library to create an interface with Erlang code, leveraging through the procedural macro system of Rust to avoid dealing with the low-level details of interfacing Erlang code with native code. This maintains a declarative approach, eliminating a lot of boilerplate code.

The final Rust code has been compiled as a dynamic library for Linux (.so), which will be loaded into the beam machine at runtime. The fundamental structs of native bindings are three opaque objects: *Context*, which corresponds to creating a thread for creating connections and statements; *Connection*, which corresponds to a connection to a particular file (and a particular database, main as default), and is intended to be referred to a particular context where it is created; and *Statement*, which corresponds to a statement relative to a connection and thus to a context. All these mentioned structs will appear to Erlang as opaque objects.

3 Erlang Cluster

We decided to make use of the Ra Library[3] for our Erlang cluster. Ra is an implementation of the Raft consensus algorithm in Erlang.

3.1 Raft Consensus Algorithm

The Raft algorithm provides a way to achieve consensus in a distributed system, ensuring all nodes in the system agree on the same data. It's often used in the management of a replicated log. Those are the main components of the Raft algorithm:

Leader Election:

- If a follower receives no communication, it becomes a candidate and starts an election.
- Candidates request votes from other nodes, a candidate votes for itself.
- The node receiving majority votes becomes the leader, this is called a *quorum* and ensures only one leader is elected even in the event of a network partition.
- When a leader is elected, it sends a heartbeat to all followers to maintain its leadership. The time in which a leader stays in power is called a *term*.

Log Replication:

- The leader accepts log entries from clients and replicates them across followers.
- Followers append entries to their logs. Each entry has an *index* and term number that identifies its position in the log and is incremented for each new entry.
- The leader waits for majority of followers to write the entries and then commits the entry, making it *durable*, eventually it will be consistent across all nodes.

So to summarize, Raft ensures that:

- There is only one leader at a time.
- Once an entry is committed, it is durable and eventually consistent across all nodes.
- The system remains available as long as the majority of nodes are available.

3.2 Erlang Node Implementation

The cluster is started from one single node, which is the leader. All other nodes when started will join the cluster as followers. When a node first comes alive it will check if it can restart the server, in case it was previously part of the cluster and it just recovered from a crash. The following graph shows the decision process for the 'ra' related operations.

[3]Ra is a Raft implementation by Team RabbitMQ: `https://github.com/rabbitmq/ra`

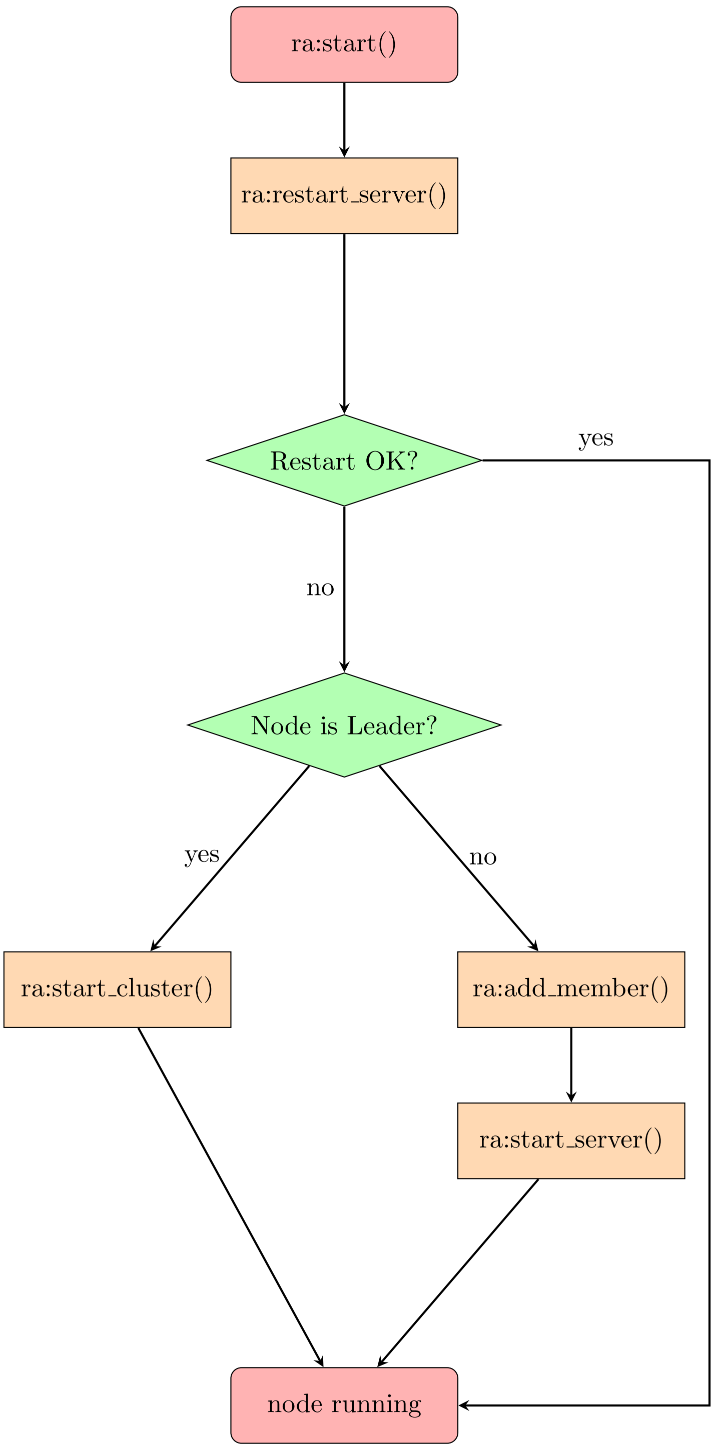


Figure 3: Decision Graph for ‘ra‘ Related Operations

The state of the system is implemented by ra as a *state machine*, on each new log entry commit the function `apply` is called with the state and the new entry as arguments, the function then returns the new state. One possible optimization is to commit to the log only the operations that change the state, instead of all the operations requested by the clients. We implemented this obtimization up to a certain point. The operations that do not change the state are just executed on the leader node. The are executed on the leader because a follower node may be not up to date with the state of the system, so it may not be able to execute the operation correctly. The following graph shows the decision process for the ‘state‘ related operations.

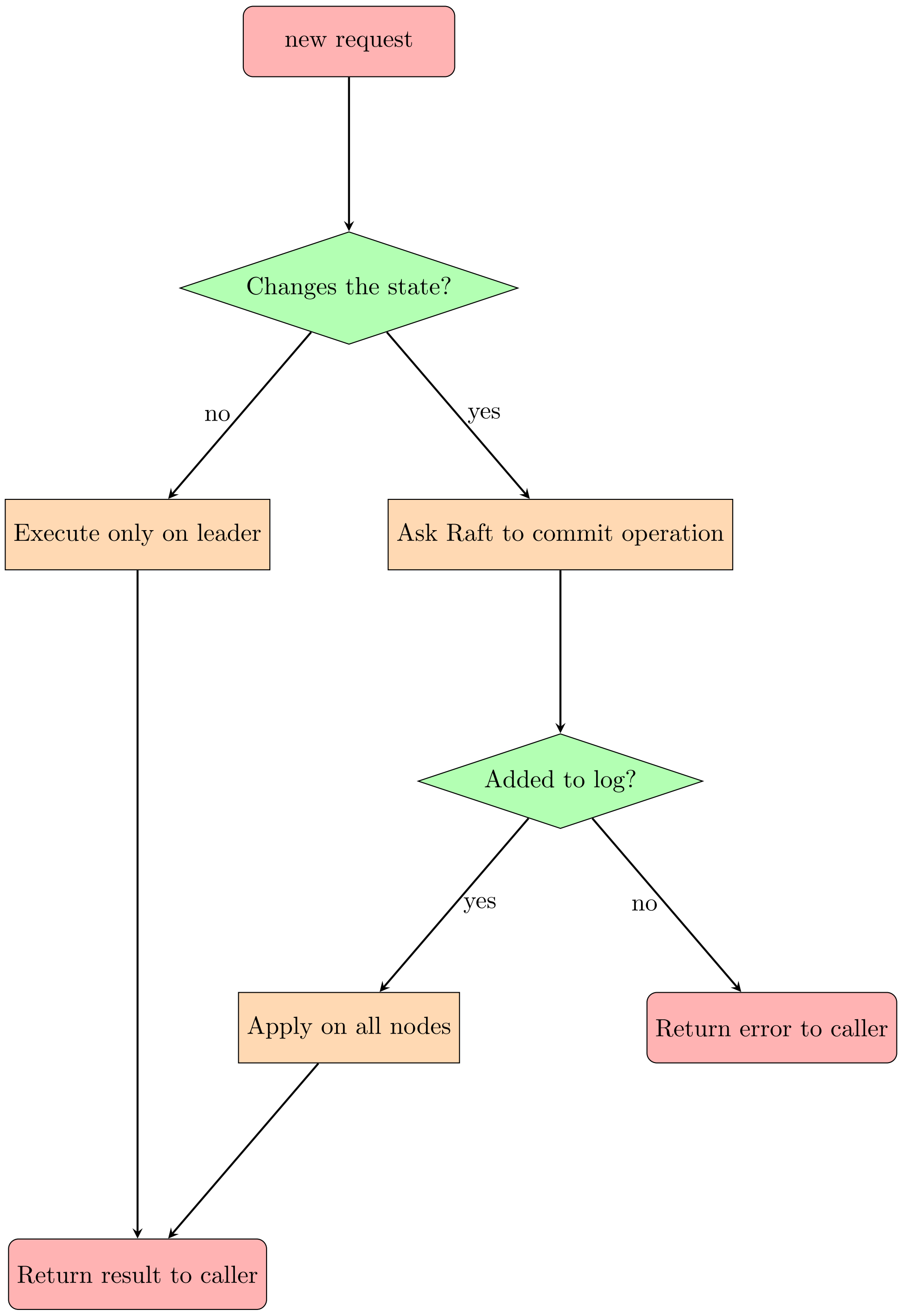


Figure 4: Decision Graph for a new request

4 Kotlin Server

Kotlin, a statically-typed programming language running on the JVM, was chosen for the JDBC driver implementation due to its syntactical simplicity and functional features, which provide the following advantages:

- **Interoperability with Java:** Kotlin's seamless integration with Java makes it an ideal choice for developing robust drivers that can operate on Linux systems.
- **Null Safety:** The language's null safety feature is instrumental in preventing common null-related issues, ensuring a more stable driver.
- **Simplicity and Maintainability:** Kotlin's concise syntax promotes readability and maintainability, enabling faster development cycles and easier codebase management.

- **Functional Programming:** Kotlin's support for functional programming concepts allows for cleaner, more efficient code when dealing with database operations.

4.1 Integration with Spring Boot and Hibernate

The JDBC driver is adeptly designed for integration with Spring Boot, utilizing the SQLite dialect for Hibernate to manage database interactions with remarkable efficiency. This integration serves as a cornerstone for developers who prioritize a robust ORM tool while minimizing the need for extensive configuration.

When the JDBC driver is used in conjunction with Spring Boot, it taps into Spring's auto-configuration mechanism, which is highly beneficial for rapid application development. Spring Boot's ability to detect and configure JDBC templates and repositories reduces the boilerplate code typically associated with database operations.

Hibernate, when integrated within this ecosystem, adds another layer of efficiency by translating Java/Kotlin classes to database tables and from Java/Kotlin data types to SQL data types. This is done while also taking care of the database connection management. The use of the SQLite dialect ensures that Hibernate is optimized for interactions with SQLite databases, allowing for lightweight deployment and local development scenarios.

Furthermore, the integration harmonizes Spring Boot's transaction management with Hibernate's unit of work principle, thereby ensuring that all database operations are part of a well-managed transaction lifecycle. This combination results in a cleaner codebase, reduced likelihood of errors, and a more straightforward approach to database operations, ultimately enhancing productivity and application robustness.

Developers benefit from the ability to focus more on business logic rather than database connectivity and transaction management. This integration also leads to better maintainability of the code and easier debugging due to the abstraction layer provided by Hibernate over the JDBC driver. In essence, the integration of the JDBC driver with Spring Boot and Hibernate encapsulates the best practices of enterprise Java development, offering a streamlined, convention-over-configuration approach that aligns well with modern software development paradigms.

4.2 Spring Boot Application

The JDBC driver is accompanied by a Spring Boot application that serves as a web-based interface for interacting with the database. This application is written in Kotlin and is designed to resemble phpMyAdmin, a popular web-based interface for MySQL databases.

The Spring Boot application is a RESTful API that exposes endpoints for performing CRUD operations on the database. It uses lombok to generate boilerplate code and reduce the amount of code needed to be written. The application also uses the Spring Data JPA framework to interact with the database, which is managed by Hibernate, as described in the previous section. The application is designed to be lightweight and portable, allowing it to be deployed on a variety of platforms. It can be run as a standalone application or as a Docker container.

Hibernate: Hibernate is an ORM (Object-Relational Mapping) tool that maps Java classes to database tables and Java data types to SQL data types. It also manages the database connection and provides transaction management. Hibernate is used by the Spring Data JPA framework to interact with the database. The application uses the SQLite dialect for Hibernate, which is optimized for SQLite databases. This allows for lightweight deployment and local development scenarios.

```kotlin
@NoArgsConstructor
@Entity
@Getter
@Setter
@Table(name = "credentials")
open class User(
    @Column(name = "username")
    private var username: String = "",
    @Column(name = "password")
    private var password: String = "",
    @Column(name = "isAdmin")
    private var isAdmin: Boolean = false,
    @Column(name = "createdAt")
    private var createdAt: Timestamp = Timestamp(System.currentTimeMillis()),
    ) {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private var id: Long? = null
}
```

Listing 1: Hibernate usage with credentials table

In the above code snippet, we can see how the User class is mapped to the credentials table in the database. The class contains the username, password, isAdmin, and createdAt fields, which are mapped to the corresponding columns in the database. The class also contains the id field, which is the primary key of the table. The @Id annotation indicates that this field is the primary key, and the @GeneratedValue annotation indicates that the value of this field is generated automatically by the database. The @Entity annotation indicates that this class is an entity, and the @Table annotation indicates the name of the table in the database. This let us to use the following code to create a new user:

```
    override fun createUser(username: String, password: String) {
        val user = User(username)
        user.setUsername(username)
        user.setPassword(password)
        entityManager.persist(user)
    }
```

Listing 2: Create a new user

As we can see, the code is much simpler than if we were to use SQL directly. This is because we are using the Spring Data JPA framework, which provides a layer of abstraction over the database. This allows us to focus on the business logic of the application instead of worrying about low-level details such as database connections and transactions. It also allows us to use object-oriented programming concepts such as inheritance and polymorphism, which are not possible with SQL. However, if we need to execute a custom SQL query, we can do so using Hibernate's native query feature:

```
override fun grantFiles(usernames: Array<String>, files: Array<String>): Boolean {
    val queryBody = usernames.flatMap {
        files.map { "SELECT ? AS username, ? AS filename" }
    }.joinToString(separator = " UNION ALL ")
    val finalQuery = """
        INSERT OR REPLACE INTO privileges (username, filename)
        SELECT * FROM ($queryBody);
    """.trimIndent()
    return cartesianProductQuery(finalQuery, usernames, files)
}
```

Listing 3: Execute a custom SQL query

In the above code snippet, we can see how we can execute a custom SQL query using Hibernate's native query feature. The query is dynamically generated based on the parameters passed to the function. This allows us to write generic code that can be reused for different use cases.

Security: The Spring Boot application is secured using Spring Security, which provides authentication and authorization mechanisms. The application uses JSON Web Tokens (JWT) for authentication, which are generated by the server and sent to the client. The client then sends the JWT with each request to the server, which verifies the token and grants access to the requested resource if the token is valid. The application also uses Spring Security's role-based authorization to restrict access to certain endpoints based on the user's role. The application has three roles:

- **Admin**:
  - Has access to all endpoints.
  - Can list and modify all file system.

    - He manages the users.

- **User**:
    - Has access to all endpoints except for the ones that modify the file system.
    - Cannot list all file system.
- **Guest**:
    - Can only access the login and sign up endpoints.

The application uses BCrypt to hash the passwords before storing them in the database.

4.3 JDBC Driver Design

The Erldb JDBC Driver is structured as follows:

- The Java Application Tier that executes the JDBC API calls.
- The REST API Client Tier within the driver that translates these calls into RESTful HTTP requests.
- The Database Server Tier that processes these requests and interacts with the Erldb database.
- Metadata Handling: The driver can retrieve metadata about the database schema, tables, and columns through specific API endpoints. This metadata is essential for applications to understand the structure of the database they are interacting with.
- Statement Optimization: The driver can optimize the execution of statements by using the step_by() function to reduce latency.
- Streaming Results: For large query results, the driver can stream data from the server to the client. This avoids loading the entire result set into memory, which can improve the performance for large datasets.
- Transactional Support: The REST API server must support transactional operations to maintain ACID (Atomicity, Consistency, Isolation, Durability) properties. The client can start, commit, or roll back transactions through the REST API.
- Fallback Mechanisms: In case the REST API server becomes unavailable, the driver can include fallback mechanisms, such as retry logic or switching to a secondary server.
- Licensing and Compliance: The driver is mindful of open-source licenses and compliance, ensuring that it can be freely used and distributed in accordance with its licensing terms.

URL Format: The URL format for the Erldb JDBC Driver is as follows:

```
jdbc:erldb://<host1>:<port1>,<host2>:<port2>,../<bucket>/<file>
```

The URL consists of the following components:

- **jdbc:erldb://** - The prefix that identifies the driver.
- **<host1>:<port1>,<host2>:<port2>,../** - The list of hosts and ports of the database nodes.
- **<bucket>** - The name of the bucket.
- **<file>** - The name of the file.

The driver will try to connect to the first host and port in the list. If it fails, it will try another random one, and so on. The driver will keep trying until it finds a host that is available. If all hosts are unavailable, the driver will throw an exception.

4.4 REST API Communication:

The Erldb driver capitalizes on REST API communication to interact with the database nodes. This design choice offers a stateless client-server architecture, which is integral to contemporary distributed systems. The RESTful approach ensures that each API call contains all the necessary information for the server to process the request, enhancing scalability by allowing servers to quickly free resources and supporting session continuity across different server instances.

Moreover, REST APIs are based on standard HTTP methods, which means that Erldb can leverage existing web infrastructure. This compatibility simplifies the integration with various services and clients, allowing for a broad ecosystem of tools for monitoring, testing, and interacting with the database layer.

The uniform interface provided by REST also facilitates a clear separation of concerns, which enhances the modularity of the system. As a result, maintenance becomes more manageable, and the scope for future upgrades or replacements is broadened without impacting the overall system.

By adhering to the principles of REST, Erldb inherently benefits from cacheability, with responses being explicitly labeled as cacheable or non-cacheable. This can significantly reduce the number of interactions needed between clients and servers, thereby reducing latency and improving the efficiency of the network.

Lastly, the RESTful nature of Erldb ensures that it can communicate effectively in a microservices architecture where services are loosely coupled and can be independently developed and deployed. This aligns Erldb with the direction of modern application development, which demands agility, flexibility, and scalability.

5 Web UI

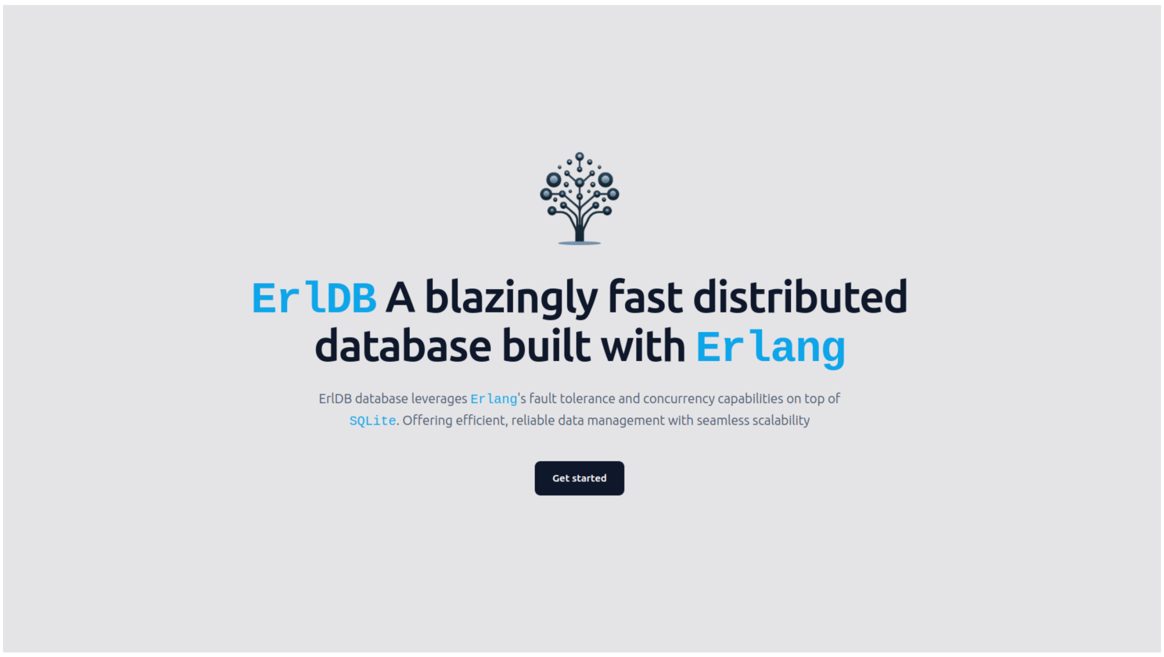
The web UI was implemented using Svelte, a modern, free, and open-source front-end component framework. Svelte stands out for its unique approach to building web interfaces. Unlike traditional frameworks that do the bulk of their work in the browser, Svelte shifts that work into a compile step that happens when you build your app. This results in highly efficient code that updates the DOM when the state of your app changes.

5.1 Features of Svelte in Web UI Development

- **Less Code**: Svelte's design allows developers to write less code, leading to fewer bugs and a more straightforward development process.
- **No Virtual DOM**: Svelte compiles to code that directly updates the DOM, which means less memory usage and improved performance.
- **Reactive by Design**: Svelte's reactivity is baked into the language, making state management simpler and more intuitive.
- **Component-based Architecture**: Like other modern frameworks, Svelte also utilizes a component-based architecture, making the code more reusable and maintainable.

5.2 Web UI Components

The web UI, built with Svelte, comprises several key components, each designed for optimal user experience and functionality.

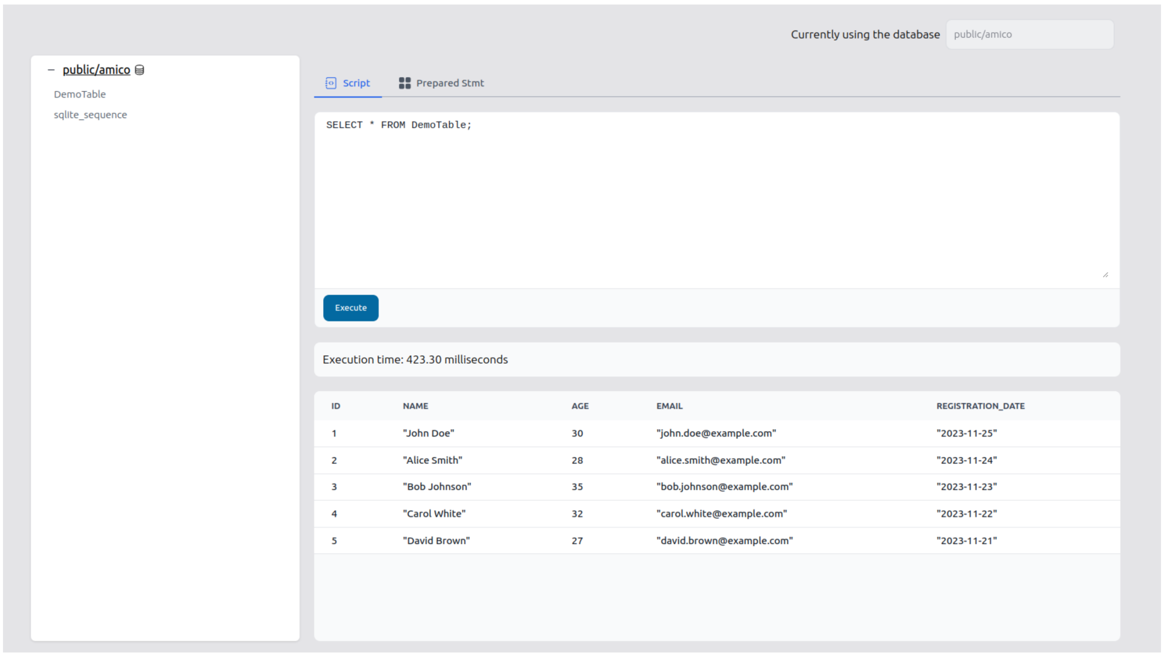


ErlDB A blazingly fast distributed database built with Erlang

ErlDB database leverages Erlang's fault tolerance and concurrency capabilities on top of SQLite. Offering efficient, reliable data management with seamless scalability

Figure 5: Home Page

The Home page is the entry point of the application. It features a clean and intuitive design, with a ”Get Started” button that leads users to the dashboard.



Currently using the database public/amico

```
SELECT * FROM DemoTable;
```

Execution time: 423.30 milliseconds

ID	NAME	AGE	EMAIL	REGISTRATION_DATE
1	"John Doe"	30	"john.doe@example.com"	"2023-11-25"
2	"Alice Smith"	28	"alice.smith@example.com"	"2023-11-24"
3	"Bob Johnson"	35	"bob.johnson@example.com"	"2023-11-23"
4	"Carol White"	32	"carol.white@example.com"	"2023-11-22"
5	"David Brown"	27	"david.brown@example.com"	"2023-11-21"

Figure 6: Query Example

The Query Example page showcases a simple query interface. It also displays table information once a user selects a table from the left panel.

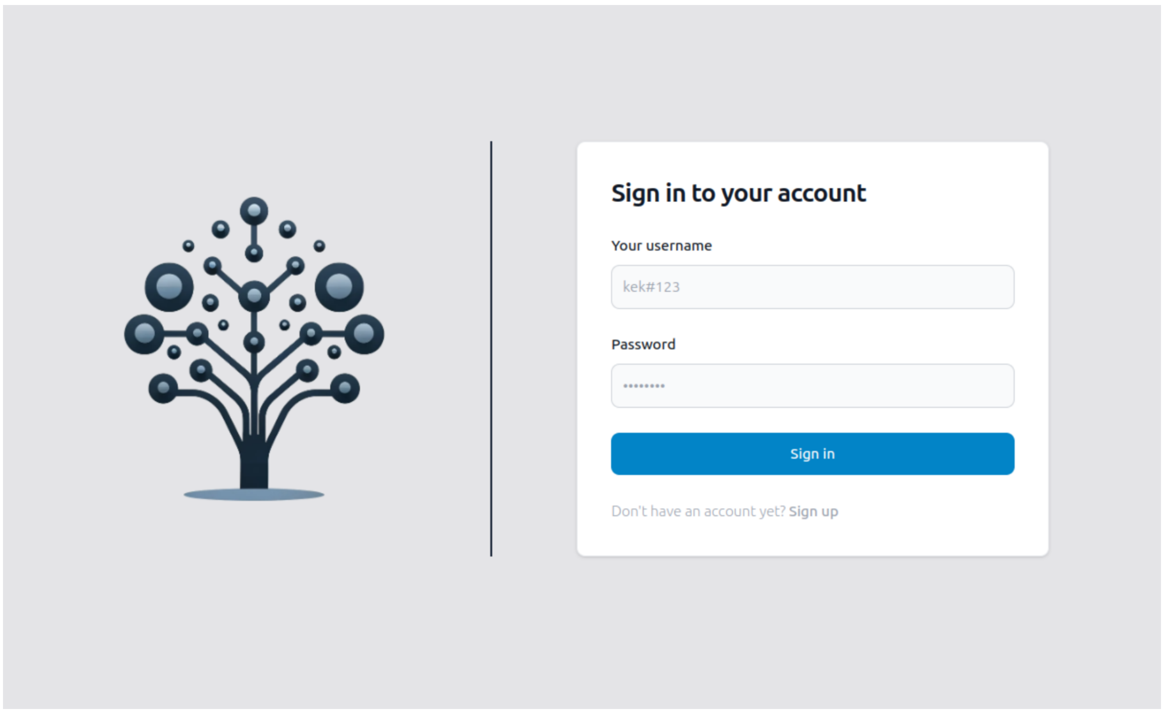
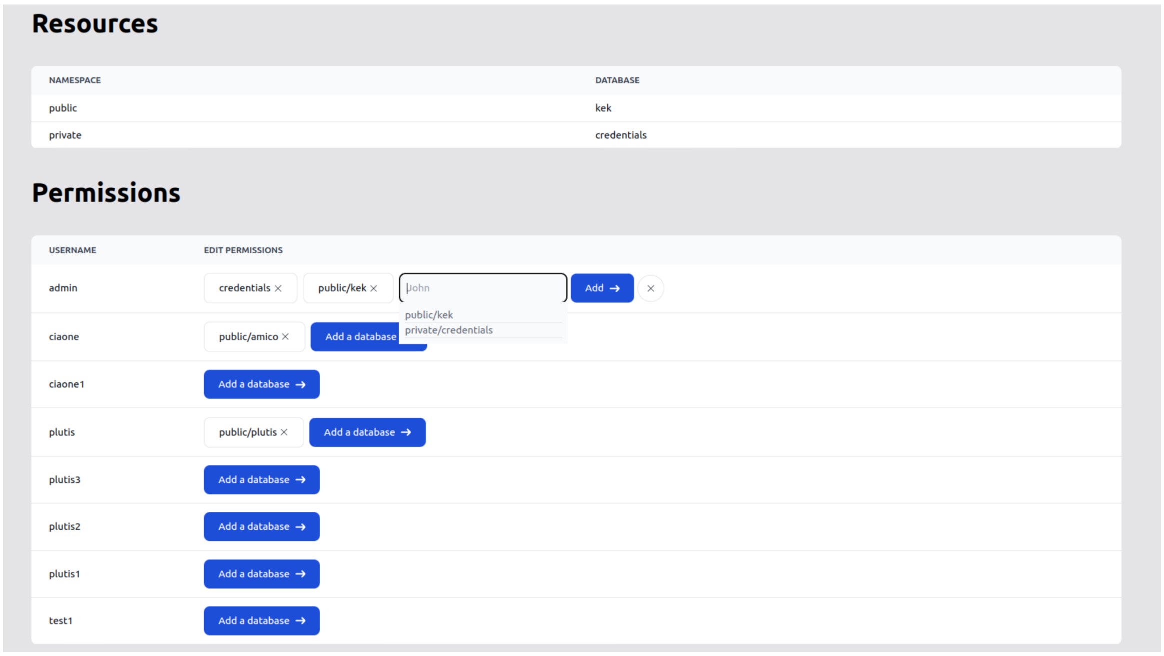


Figure 7: Login Page

The Login interface provides secure and user-friendly access to the application.



Resources

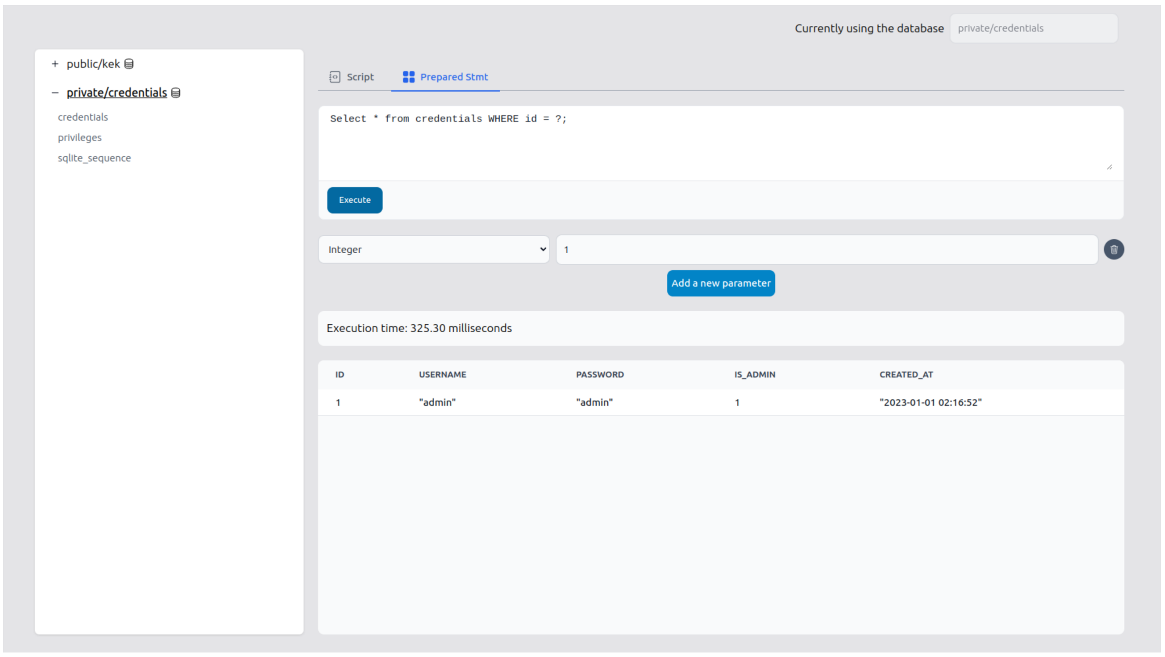
NAMESPACE	DATABASE
public	kek
private	credentials

Permissions

USERNAME	EDIT PERMISSIONS
admin	credentials × public/kek × John Add → × (public/kek, private/credentials)
ciaone	public/amico × Add a database
ciaone1	Add a database →
plutis	public/plutis × Add a database →
plutis3	Add a database →
plutis2	Add a database →
plutis1	Add a database →
test1	Add a database →

Figure 8: Admin Dashboard

The Admin Dashboard is a crucial part of the application, offering database management, access control, and permission settings. It visually represents databases within namespaces and provides an editable user list.



```
Select * from credentials WHERE id = ?;
```

Execution time: 325.30 milliseconds

ID	USERNAME	PASSWORD	IS_ADMIN	CREATED_AT
1	"admin"	"admin"	1	"2023-01-01 02:16:52"

Figure 9: Prepared Statement Interface

The Prepared Statement Interface allows users to execute database queries efficiently. It supports queries with placeholders, represented by the '?' symbol, and allows for the passing of arguments through parameter inputs.

5.3 Conclusion

Utilizing Svelte for this web UI has enabled the creation of a highly efficient, user-friendly interface with modern features. Its reactive nature and component-based architecture make it an ideal choice for developing responsive web applications.

6 Conclusions

In conclusion, the DSMT Project successfully develops a robust, distributed SQLite database system that emphasizes redundancy and fault tolerance, leveraging the strengths of Erlang/OTP, Java, and web technologies. This innovative system not only ensures high availability and consistency across the network but also provides a user-friendly interface for efficient data management, making it an ideal solution for applications requiring reliable and accessible database operations.