# Documentation - TI Project

## Setup

1. Have an OracleDB running with the port it is running on forwarded
2. Add a user in the database meant for the API to connect as, give permissions accordingly
3. In api/src/main/resources/application.properties, change the db.url, db.username, db.password according to the database url, username and password. Change server.port to select the port you'd like the API to run on.
4. *Run the API and make sure that port is forwarded.
5. To import client functions into your program, for C++, include ApiClient.h and json.hpp to use nlohmann::json. When compiling, use the -lcurl tag and -std=c++11. For Java, bring the ApiClient class into your project, and add the dependency jackson-databind.

   *: When the code and properties for the API are finalized, run "mvn clean package". If there is an error about the maven-shade plugin, disregard it. Take the jar at api/target/database-connection-service-1.0-SNAPSHOT.jar. The API and all its dependencies will run from that one jar file.

---

### 1. Overview

This program provides a system for interacting with a relational database through a RESTful API and client implementations in both Java and C++. It simplifies database operations by abstracting SQL into API endpoints and allows users to execute queries, manage tables, and handle records programmatically.

---

## 2. Class Descriptions

---

### 2.1. `DatabaseController`

**Package:** `org.example.DatabaseAPI.controller`

- **Purpose:**
  Acts as the primary controller for exposing RESTful endpoints for database operations.
- **Key Dependencies:**
  1. `DBHandler`: Abstract interface for database interaction.
  2. `OracleDBHandler`: Oracle-specific implementation of `DBHandler`.
- **Key Methods:**
  1. **`execQuery(Map<String, String> request)`**
     - **Endpoint:** `POST /api/execQuery`

        - **Purpose:** Executes a SQL query and returns the result.
        - **Input:** JSON request body with the query string.
        - **Output:** Response entity with the query result or error message.
    2. **`createTable(String sqlStr)`**
        - **Endpoint:** `POST /api/createTable`
        - **Purpose:** Executes a SQL statement to create a new table.
        - **Input:** SQL string for table creation.
        - **Output:** Response entity with the status of the operation.
    3. **`listTables()`**
        - **Endpoint:** `GET /api/listTables`
        - **Purpose:** Lists all database tables and their schema.
        - **Input:** None.
        - **Output:** Response entity containing table metadata.
    4. **`insert(Map<String, Object> payload)`**
        - **Endpoint:** `POST /api/insert`
        - **Purpose:** Inserts records into a specified table.
        - **Input:** JSON object with `tableName` and `values`.
        - **Output:** Response entity with the status of the operation.
    5. **`delete(Map<String, Object> payload)`**
        - **Endpoint:** `POST /api/delete`
        - **Purpose:** Deletes records matching conditions from a specified table.
        - **Input:** JSON object with `tableName`, `columns`, and `values`.
        - **Output:** Response entity with the status of the operation.
    6. **`select(Map<String, Object> payload)`**
        - **Endpoint:** `POST /api/select`
        - **Purpose:** Retrieves records from a table based on conditions.
        - **Input:** JSON object with `tableName`, `columns`, `whereClause`, and `params`.
        - **Output:** Response entity with query results.

---

### 2.2. `DatabaseConfig`

**Package:** `org.example.config`

- **Purpose:**
  Configures and provides a bean for the `OracleDBHandler` class, allowing dependency injection in the `DatabaseController`.
- **Key Methods:**
    1. **`OracleDBHandler(String url, String user, String password)`**

        - **Purpose:** Creates an instance of `OracleDBHandler` using database connection properties from the `application.properties` file.
        - **Usage:** Injected into `DatabaseController`.

---

### 2.3. `ApiApp`

**Package:** `org.example`

- **Purpose:**
  Entry point for the Spring Boot application.
- **Key Method:**
    - **`main(String[] args)`**
        - Bootstraps the application using `SpringApplication.run()`.

---

### 2.4. `DBHandler`

**Package:** `org.example`

- **Type:** Abstract class
- **Purpose:**
  Provides a base implementation for interacting with a relational database.
- **Key Attributes:**
    1. `url`: Database connection URL.
    2. `username`: Username for database authentication.
    3. `password`: Password for database authentication.
- **Key Methods:**
    1. **`connect()`**
        - **Purpose:** Establishes a connection to the database.
        - **Usage:** Used internally by all database operations.
    2. **`execQuery(String query)`**
        - **Purpose:** Executes a raw SQL query and returns the result.
        - **Logic:** Differentiates between `SELECT` and other SQL queries.
    3. **`createTable(String sqlStr)`**
        - **Purpose:** Executes a SQL statement to create a table.
    4. **`listTables()`** (Abstract)
        - **Purpose:** Lists all database tables and their metadata.
        - **To Be Implemented:** By subclasses for database-specific logic.
    5. **`insert(String tableName, List<Object> values)`** (Abstract)
        - **Purpose:** Inserts records into a table.

        - **To Be Implemented:** By subclasses to validate data and handle database-specific operations.
    6. `delete(String tableName, List<String> columns, List<Object> values)` (Abstract)
        - **Purpose:** Deletes records matching conditions from a table.
        - **To Be Implemented:** By subclasses for database-specific logic.
    7. `select(String tableName, List<String> columns, String whereClause, List<Object> params)`
        - **Purpose:** Retrieves records from a table with filtering conditions.
        - **Logic:** Constructs the SQL query dynamically and sets parameters.
    8. `toJavaLikeType(String dataType, int dataLength)` (Abstract)
        - **Purpose:** Converts SQL data types to Java-like types for metadata representation.
    9. `generateExampleValue(String dataType, int dataLength)` (Abstract)
        - **Purpose:** Generates example values for database table columns.

### 2.5. `OracleDBHandler`

**Package:** `org.example`

- **Purpose:**
  Implements Oracle-specific logic for database operations, extending the functionality of the abstract `DBHandler` class.
- **Key Methods:**
    1. `listTables()`
        - **Purpose:** Retrieves metadata about all tables in the Oracle database, including column names, data types, and example values.
        - **Logic:**
            - Executes a query on `user_tab_columns` to gather table schema information.
            - Formats the output with example values and column metadata.
        - **Output:** A `Result` object containing a formatted schema description or an error message.
    2. `toJavaLikeType(String dataType, int dataLength)`
        - **Purpose:** Maps Oracle SQL data types to equivalent Java-like data types.
        - **Logic:**
            - `VARCHAR2` and `CHAR` map to `String`.
            - `NUMBER` maps to `int`, `long`, or `double` depending on `dataLength`.
            - `DATE` maps to `LocalDate`.

3. **`generateExampleValue(String dataType, int dataLength)`**
    - **Purpose:** Generates example values for table columns based on data type.
    - **Logic:**
        - `VARCHAR2` and `CHAR` produce `"example_string"`.
        - `NUMBER` produces `123` or `1234567890` depending on length.
        - `DATE` produces `LocalDate.now()`.
4. **`insert(String tableName, List<Object> values)`**
    - **Purpose:** Inserts a row into a specified table.
    - **Logic:**
        - Validates table existence and column data types.
        - Prepares an `INSERT INTO` statement with placeholders for values.
        - Executes the statement and returns a `Result` indicating success or failure.
5. **`delete(String tableName, List<String> columns, List<Object> values)`**
    - **Purpose:** Deletes rows from a table based on specified conditions.
    - **Logic:**
        - Validates table existence and column names.
        - Constructs a `DELETE FROM` statement with a dynamic `WHERE` clause.
        - Executes the statement and returns a `Result` indicating success or failure.
6. **`isValueCompatibleWithType(Object value, String expectedDataType)`**
    - **Purpose:** Verifies if a given value matches the expected SQL data type.
    - **Logic:**
        - Validates `VARCHAR2`/`CHAR` as `String`, `NUMBER` as `Number`, and `DATE` as `java.sql.Date` or `LocalDate`.

---

### 2.6. `Result`

**Package:** `org.example`

- **Purpose:**
  Encapsulates the outcome of a database operation, including status, message, and optionally, data.
- **Key Attributes:**
    1. `status` (String): Indicates success or failure (`success` or `error`).

2. `message` (String): Describes the operation result.
3. `data` (String, optional): Contains additional data, such as query results.

- **Key Methods:**
  1. **`Result(String status, String message)`**
     - Constructor for operations without data.
  2. **`Result(String status, String message, String data)`**
     - Constructor for operations with data.
  3. **`getStatus()`**
     - Retrieves the operation status.
  4. **`getMessage()`**
     - Retrieves the operation message.
  5. **`getData()`**
     - Retrieves the operation data, if available.

---

### 2.7. `DatabaseControllerTest`

**Package:** `org.example.DatabaseAPI.controller`

- **Purpose:**
  Tests the `DatabaseController` class to validate the functionality of API endpoints using a stubbed `OracleDBHandler`.
- **Key Tests:**
  - **`testExecQuery_Success()`**
    - Validates successful execution of a SQL query.
  - **`testExecQuery_MissingQuery()`**
    - Ensures proper error handling for missing or invalid queries.
  - **`testCreateTable_Success()`**
    - Verifies table creation logic.
  - **`testListTables_Success()`**
    - Validates retrieval of table metadata.
  - **`testInsert_Success()`**
    - Tests data insertion functionality.
  - **`testDelete_Success()`**
    - Validates record deletion logic.
  - **`testSelect_Success()`**
    - Verifies data retrieval from a table.
- **Stub Implementation (`OracleDBHandlerStub`):**
  - Overrides methods in `OracleDBHandler` to return predefined results for testing.

## 3. Build Configuration

---

### `pom.xml`

**Purpose:**
Defines the Maven build configuration for the project.

- **Key Sections:**
  1. **Parent Definition:**
     - Uses `spring-boot-starter-parent` for dependency management.
  2. **Dependencies:**
     - **Spring Boot Web:** For REST API development.
     - **Oracle JDBC Driver:** For Oracle database connectivity.
     - **JUnit 5:** For testing.
     - **Spring Boot Test:** For Spring-specific test utilities.
     - **H2 Database:** For in-memory testing.
  3. **Plugins:**
     - **Spring Boot Maven Plugin:** For building and running the application.
     - **Maven Compiler Plugin:** Configured for Java 11.

## 4. C++ Client Functions Overview

The C++ implementation provides a client for interacting with the RESTful API defined earlier. It uses `libcurl` for HTTP requests and `nlohmann::json` for JSON parsing and manipulation. The client allows execution of SQL queries, table creation, listing tables, inserting data, deleting data, and selecting data.

---

### 4.1. `ApiClient.cpp`

**Purpose:**
Implements the methods defined in the `ApiClient` class for sending requests to the API and processing the responses.

**Key Components:**

1. `sendRequest`
   - **Purpose:**
     Handles HTTP communication with the API server. Configures `libcurl` for sending requests, including setting headers and handling response data.
   - **Input:**
     - `url` (String): Endpoint URL.
     - `method` (String): HTTP method (`POST`, `GET`).

        - `body` (String, optional): Request payload for `POST` requests.
    - **Output:**
        - Response body as a string. Returns an empty string on failure.
2. **`execQuery`**
    - **Purpose:**
      Executes a SQL query by calling the `/api/execQuery` endpoint.
    - **Input:**
        - `query` (String): SQL query string.
    - **Output:**
        - Parsed JSON object containing the query results or an error message.
    - **Example:** execQuery("SELECT * FROM PEOPLE")
3. **`createTable`**
    - **Purpose:**
      Creates a table in the database using the `/api/createTable` endpoint.
    - **Input:**
        - `tableSql` (String): SQL statement for table creation.
    - **Output:**
        - `true` if successful, `false` otherwise.
    - **Example:** createTable("CREATE TABLE test (id INT, name VARCHAR(50))")
4. **`listTables`**
    - **Purpose:**
      Retrieves metadata about all tables in the database by calling the `/api/listTables` endpoint.
    - **Output:**
        - Parsed JSON object containing table metadata or an error message.
    - **Example:** listTables()
5. **`insert`**
    - **Purpose:**
      Inserts a record into a table using the `/api/insert` endpoint.
    - **Input:**
        - `tableName` (String): Name of the target table.
        - `values` (String): JSON array representing the values to be inserted.
    - **Output:**
        - `true` if successful, `false` otherwise.
    - **Example:** insert("PEOPLE", "[10, \"Doe\", \"John\", \"Renner Rd\", \"Dallas\"]")
6. **`deleteData`**
    - **Purpose:**
      Deletes records from a table using the `/api/delete` endpoint.
    - **Input:**
        - `tableName` (String): Name of the target table.
        - `columns` (String): JSON array of column names for the `WHERE` clause.

        - `values` (String): JSON array of corresponding values.
    - **Output:**
        - `true` if successful, `false` otherwise.
    - **Example:** deleteData("PEOPLE", "[\"FirstName\"]", "[\"John\"]")
7. **`select`**
    - **Purpose:**
      Retrieves records from a table using the `/api/select` endpoint.
    - **Input:**
        - `tableName` (String): Name of the target table.
        - `columns` (String): JSON array of column names to retrieve.
        - `whereClause` (String): SQL `WHERE` clause string.
        - `params` (String): JSON array of parameters for the `WHERE` clause.
    - **Output:**
        - Parsed JSON object containing query results or an error message.
    - **Example:** select("PEOPLE", "[\"firstname\", \"address\"]", "firstname LIKE ?", "[\"A%\"]")

**Helper Functions:**

- **`WriteCallback`**
    - **Purpose:**
      Processes HTTP response data and appends it to a string.
    - **Usage:**
      Used as the callback function for `libcurl`.

---

## 4.2. `ApiClient.h`

**Purpose:**
Declares the `ApiClient` class and its methods.

**Key Components:**

1. **Attributes:**
    - `baseUrl` (String): Base URL for the API server.
2. **Methods:**
    - **`execQuery`**: Executes a SQL query.
    - **`createTable`**: Creates a table in the database.
    - **`listTables`**: Retrieves metadata about tables.
    - **`insert`**: Inserts data into a table.
    - **`deleteData`**: Deletes records from a table.

- `select`: Selects records from a table.
- `sendRequest`: Handles HTTP communication.

## 5. Java Client Functions for API Interaction

---

### 5.1. `ApiClient.java`

**Purpose:**
Provides a Java-based client for interacting with the RESTful API. It uses the `HttpClient` library for sending HTTP requests and `ObjectMapper` from Jackson for JSON parsing and serialization.

---

**Key Methods:**

1. **`execQuery(String query)`**
   - **Description:** Executes a SQL query using the `/api/execQuery` endpoint.
   - **Input:** SQL query string.
   - **Output:** A `JsonNode` containing the query results or an error message.
   - **Example:** execQuery("SELECT * FROM PEOPLE")
2. **`createTable(String tableSql)`**
   - **Description:** Creates a table using the `/api/createTable` endpoint.
   - **Input:** SQL statement for table creation.
   - **Output:** A boolean indicating success or failure.
   - **Example:** createTable("CREATE TABLE test (id INT, name VARCHAR(50))")
3. **`listTables()`**
   - **Description:** Retrieves a list of all database tables and their metadata using the `/api/listTables` endpoint.
   - **Output:** A `JsonNode` containing table metadata or an error message.
   - **Example:** listTables()
4. **`insert(String tableName, String values)`**
   - **Description:** Inserts data into a specified table using the `/api/insert` endpoint.
   - **Input:**
     - `tableName`: Name of the target table.
     - `values`: JSON array representing the data to insert.
   - **Output:** A boolean indicating success or failure.
   - **Example:** insert("PEOPLE", "[10, \"Doe\", \"John\", \"Renner Rd\", \"Dallas\"]")
5. **`deleteData(String tableName, String columns, String values)`**

- **Description:** Deletes records from a table using the `/api/delete` endpoint.
- **Input:**
  - `tableName`: Name of the target table.
  - `columns`: JSON array of column names for the `WHERE` clause.
  - `values`: JSON array of values corresponding to the columns.
- **Output:** A boolean indicating success or failure.
- **Example:** deleteData("PEOPLE", "[\"FirstName\"]", "[\"John\"]")

6. **`select(String tableName, String columns, String whereClause, String params)`**
   - **Description:** Retrieves data from a table using the `/api/select` endpoint.
   - **Input:**
     - `tableName`: Name of the target table.
     - `columns`: JSON array of columns to retrieve.
     - `whereClause`: SQL `WHERE` clause as a string.
     - `params`: JSON array of parameters for the `WHERE` clause.
   - **Output:** A `JsonNode` containing the query results or an error message.
   - **Example:** select("PEOPLE", "[\"firstname\", \"address\"]", "firstname LIKE ?", "[\"A%\"]")

**Helper Methods:**

- **`sendRequest(HttpRequest request)`**
  - **Purpose:** Sends HTTP requests and processes responses.
  - **Input:** An `HttpRequest` object.
  - **Output:** A `JsonNode` containing the response data or `null` if an error occurs.

---

### 5.3. `pom.xml`

**Purpose:**
Defines the Maven configuration for the Java client project.

---

**Key Sections:**

1. **Dependencies:**
   - **`jackson-databind`:** For JSON parsing and serialization.
2. **Build Plugins:**
   - **Maven Compiler Plugin:** Ensures compatibility with Java 11.

---

## Usage Notes

1. **Setup:**
   - Configure the `baseUrl` in `ApiClient` to point to the API server (e.g., `http://localhost:8080`).
2. **Execution:**
   - Use the `Main` class to test the functionality of the client methods.
   - Replace SQL queries, table names, and conditions to adapt to your database schema.
3. **Error Handling:**
   - Ensure valid JSON formatting for input data (`values`, `columns`, etc.).
   - Catch exceptions in `sendRequest` and log error messages for debugging.