**Chat2DB Testing and Debugging**

**Start Date: February 4, 2024**

**Prepared by: Weixiao Liang/ Yifan He**

**Class: SWE 261P LEC A: SW TEST & DEBUG**

# Part 1: Testing and Partitioning

## Introduction

### Purpose:

Chat2DB is a multi-database client tool that has integrated the AIGC. It can convert natural language into SQL. It can also convert SQL into natural language and provide optimization suggestions for SQL to greatly enhance the efficiency of developers. Chat2DB supports various AI models and databases. With the help of AI, even non-SQL business operators in the future can use it to quickly query business data and generate reports.

### Framework:

The project consists of a front-end UI which is shown as web pages, as well as a back-end server. Users could install and run the project on Windows, Mac, Linux and web pages.

The project mainly uses **Electron + JavaScript + Java in order to support web and desktop applications. T**he primary programming languages used in the project are Java (62.0%), TypeScript (28.2%), HTML (5.3%), Less (3.8%), JavaScript (0.6%), and Shell (0.1%). Among all of them, there are 834 Java classes, which take up to 37399 lines of code.

## Deployment

### Run the Server

1. Install Maben
2. In the terminal enter the server package: cd chat2db-server
3. Use Maven to clean and install the project: mvn clean install
4. Enter the application directory: cd chat2db-server-start/target/
5. Run the application with **APIkey** argument: java -jar -Dloader.path=./lib -Dchatgpt.apiKey=xxxxx chat2db-server-start.jar
6. You can also run the server by launching the Spring boot Application, which is auto configured by IDE.

### Run the Client:

1. In the terminal enter the client package: cd chat2db-client
2. Install Node.js (including **npm**)
3. Use **npm** to install Yarn: npm install -g yarn
4. Use Yarn to resolve dependencies and download packages: yarn
5. Run the client: yarn run start:web

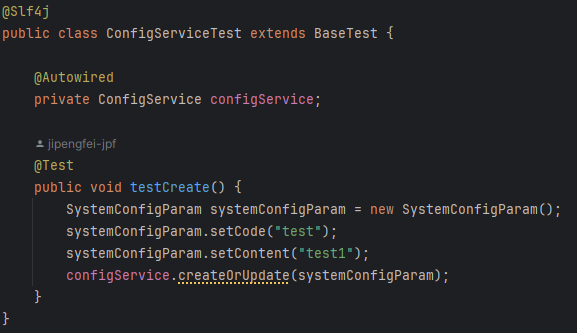
### Original Testcases

Most test cases exist in the chat2db-server-start module and chat2db-server-start module. Among them, the Junit framework and Spring Boot test framework are used. As an abstract class, BaseTest.java has used the “**@SpringBootTest**” annotation from the Spring Boot test framework, and some other classes extend the **BaseTest** class and also use the “@Test” annotation from the Junit framework.

To test the whole project, since the project uses Maven for management, we can either use “mvn test” command in the terminal or run the “test” lifecycle in our IDE. In order to run a specific test case, we can either use the “mvn -Dtest=ClassName test” command in the terminal or simply run or debug the class in IDE.

## Test Partitioning and Cases

### Existing Test Cases



*Image of existing test case* ***ConfigServiceTest***

Objective: This test class is designed to verify the functionality of **ConfigService**, specifically focusing on the creation and updating of system configuration parameters.

Test Case: **testCreate**

* **Description:** Validates that the **ConfigService** correctly handles the creation or update of a given system configuration parameter.
* **Methodology:**
  + A **SystemConfigParam** object is instantiated with predefined values.
  + The **createOrUpdate** method of the **ConfigService** is invoked with this object.
* **Expected Outcome:** The test implicitly verifies the operation by checking for the absence of exceptions. Additional assertions can be added to confirm that the configuration parameter is correctly persisted or updated in the system.

### Partition Testing Concept

Partition testing, also known as equivalence partitioning, is a testing technique that divides the input data of a software application into partitions of equivalent data from which test cases can be derived. By testing a single representative from each partition, it is assumed that all equivalent values within that partition will produce similar results, thus minimizing the total number of tests that must be conducted while ensuring adequate coverage across the range of inputs.

### Feature Selection for Partitioning

For this project, we selected the database connection functionality provided by the **IDriverManager** interface for partition testing. This functionality is critical because it enables the software to interact with a database, which is a core component of many applications. Proper testing ensures that the software can reliably connect to databases under various configurations.

### Specifying Partitions and Boundaries

We identified several partitions based on the database URL configurations, specifically focusing on variations in the port number. These partitions include:

* **Correct Port:** The port number is correct and matches the database server's configuration.
* **Wrong Port:** The port number is incorrect, representing two scenarios: a non-existent port (3307), an invalid port designation ("x"), a port with a number that has exceeded the valid range (65536), and a port with a reserved port number by the system (1023).
* **Null Port:** The port information is omitted, implying that the default port should be used.
* **Negative Port Number:** The port uses a negative port number (-1), implying that the default port should be used.

For each partition, we chose representative values that exemplify typical scenarios within that partition. For example, the correct port partition uses the standard MySQL port number (3306), while the wrong port partition tests both an incorrect numerical port and an invalid character.

### New JUnit Test Cases

**Test Case 1: Connection with Correct Port**

* **Objective**: Verify that the software can establish a database connection using the correct port.
* **Method**: Utilize the **DriverConfig** to set up a connection with the standard MySQL port (3306).
* **Expected Result:** The test should pass, indicating that the connection is not null and no exceptions are thrown.



*The code of test case 1*

**Test Case 2: Connection with Wrong Port (Numeric)**

* **Objective**: Test the software's response to an incorrect, but numeric, port (3307).
* **Method**: Configure **DriverConfig** with a non-existent port number.
* **Expected Result**: The test should throw an **SQLException**, indicating a failure to connect.



*The code of test case 2*

**Test Case 3: Connection with Wrong Port (Invalid)**

* **Objective**: Test the software's response to an invalid port designation ("x").
* **Method**: Configure **DriverConfig** with an invalid port representation.
* **Expected Result**: The test should throw an **SQLException**.



*The code of test case 3*

**Test Case 4: Connection with Wrong Port (Numeric)**

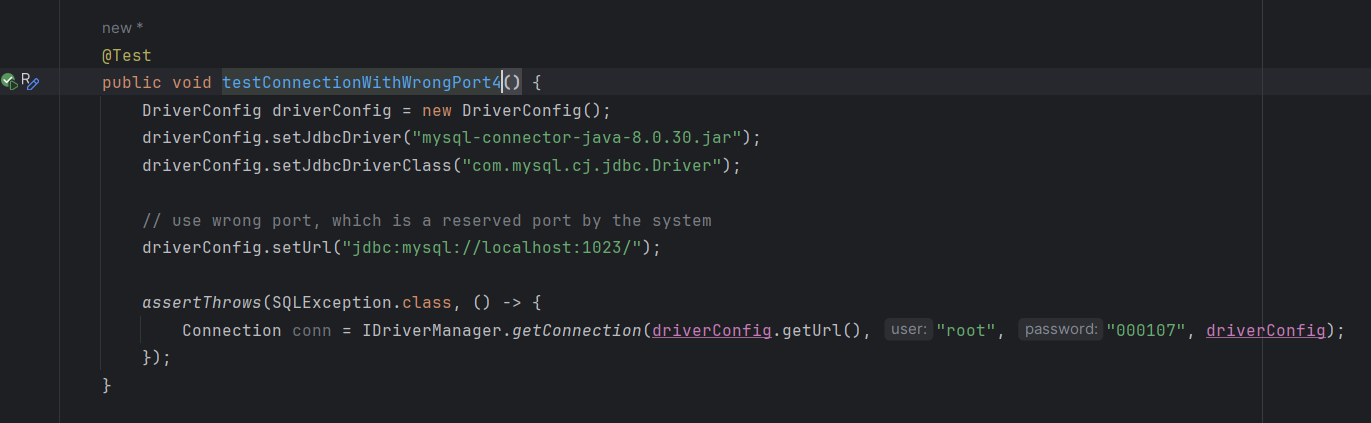
* **Objective**: Test the software's response to an invalid port designation (65536).
* **Method**: Configure **DriverConfig** with an invalid port representation.
* **Expected Result**: The test should throw an **SQLException**.



*The code of test case 4*

**Test Case 5: Connection with Wrong Port (Numeric)**

* **Objective**: Test the software's response to an invalid port designation (1023).
* **Method**: Configure **DriverConfig** with an invalid port representation.
* **Expected Result**: The test should throw an **SQLException**.



*The code of test case 5*

**Test Case 6: Connection with Negative Port (Numeric)**

* **Objective**: Test the software's response to an invalid port designation (-1).
* **Method**: Omit the port information in the **DriverConfig**.
* **Expected Result**: The test should pass, indicating a successful connection using the default port.



*The code of test case 6*

**Test Case 7: Connection with Null Port**

* **Objective**: Verify the software's ability to handle a missing port, implying the use of the default port.
* **Method**: Omit the port information in the **DriverConfig**.
* **Expected Result**: The test should pass, indicating a successful connection using the default port.



*The code of test case 7*

## Conclusion

In conclusion, the testing efforts detailed in this document represent a comprehensive approach to verifying the database connection functionality of the Chat2DB software. Through the use of partition testing, we have systematically explored the software's ability to handle various configurations of database connection URLs, specifically focusing on correct, incorrect, and absent port numbers. These tests are crucial for ensuring that Chat2DB can reliably connect to databases under diverse conditions, thereby supporting its goal of enhancing developer efficiency through AI-guided SQL generation and optimization.

Our testing has confirmed that the software behaves as expected when provided with correct port information and appropriately handles errors when confronted with incorrect or missing port data.

# Part 2: Functional Testing and Finite State Machines

## Finite State Machine

Finite State Machine is a set of states and a set of transitions. After some events happen, the transition from one state to another may happen. And some actions could occur due to the transition. FSM-based could cover more scenarios, including edge cases, thus decreasing the risk that a function is not well tested.

Also, the framework of FSM helps to understand the application behaviors clearly

## Finite State Machine in Chat2DB

The Finite State Machine could be applied to the user interface of Console and Output page in Chat2DB.

There are 4 states of the UI:

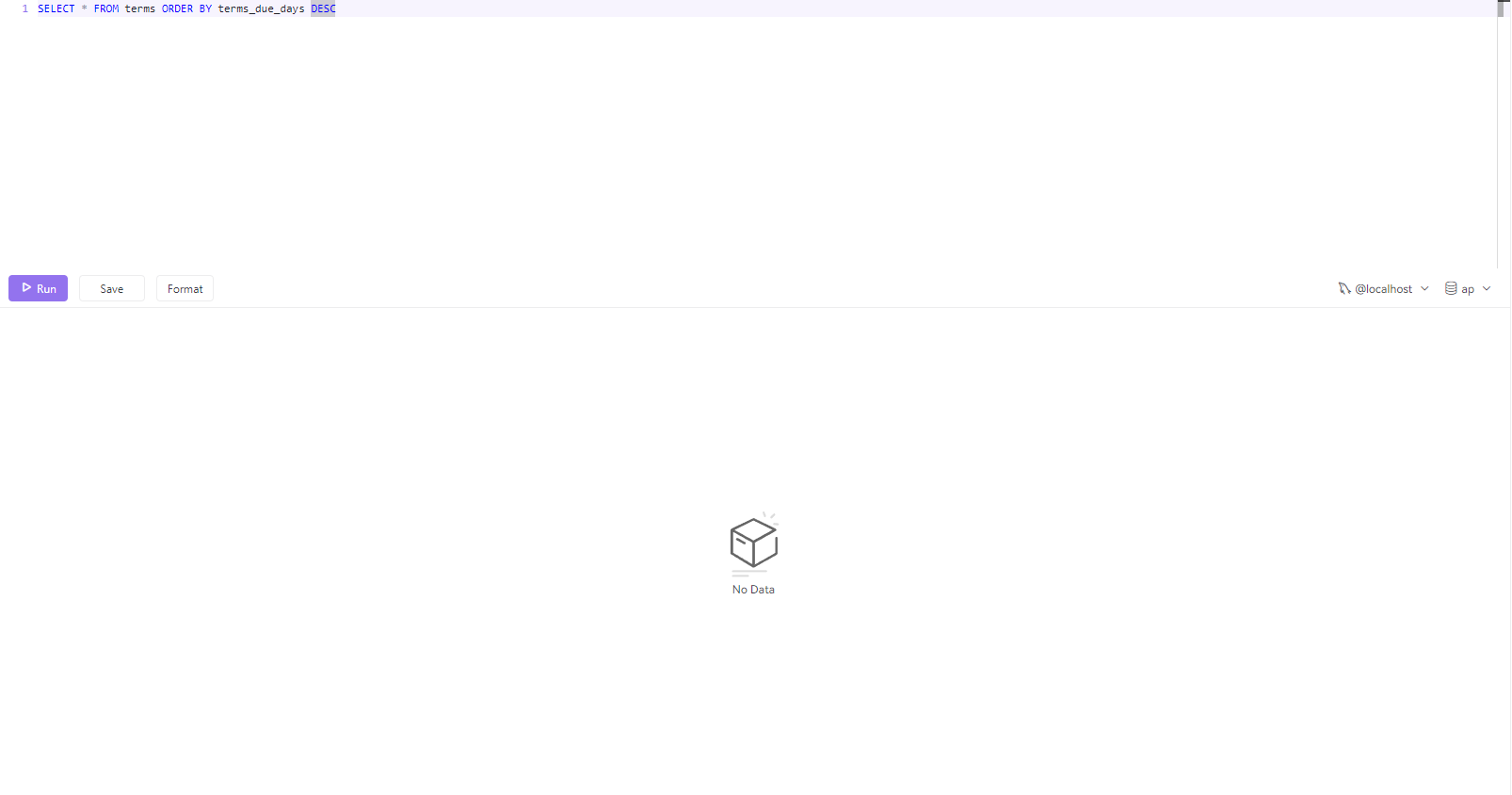
1. **Initial State**: This is the state when the SQL query is inputted as a single line.
2. **Formatted State**: After the "format" button is clicked, the SQL query is formatted into multiple lines.
3. **Saved State**: If the "save" button is clicked, the current console of the query is saved.
4. **Executed State**: When the "run" button is pressed, the SQL query is executed, and the results are displayed.

Here’s a table of states and their transitions into a new state:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| States | Description | Input: Press Button Run | Input: Press Button Save | Input: Press Button Format |
| Initial State | The SQL query is inputted as a single-line | **Executed State** | **Saved State** | **Formatted State** |
| Formatted State | The SQL query is formatted into multiple lines | **Executed State** | **Saved State** | **Formatted State** |
| Executed State | The SQL query is executed, and the results are displayed. | **Executed State** | **Saved State** | **Formatted State** |
| Saved State | The query console is saved | **Executed State** | **Saved State** | **Formatted State** |

Table of states and their transitions into a new state

Some Screenshots for each of the states:



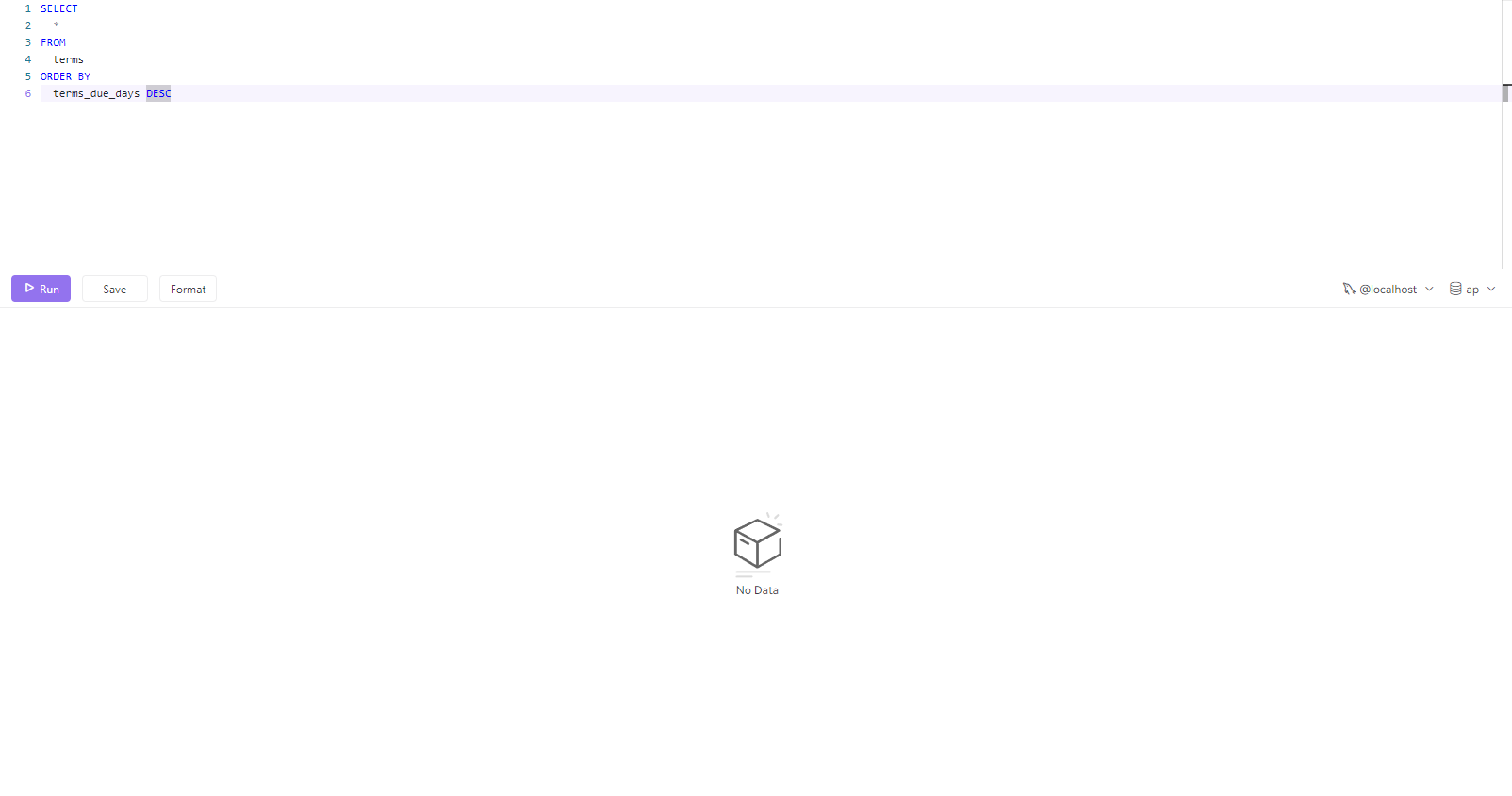
Initial State



Executed State

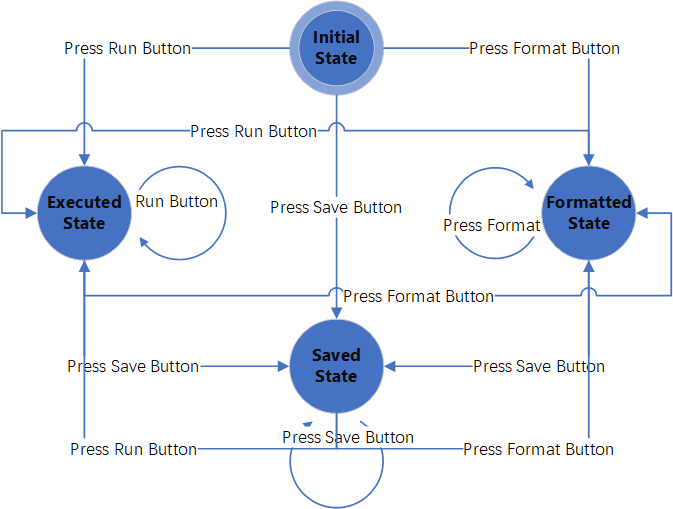


Saved State



Formatted State

Here’s a graph of the model of the Finite State Machine:



Finite State Machine Model in Chat2DB

## New Test Cases

### Test “Press Run Button” JUnit Test Cases

The Run Button will execute the SQL query.

**Test Case 1: testExecuteQuery**

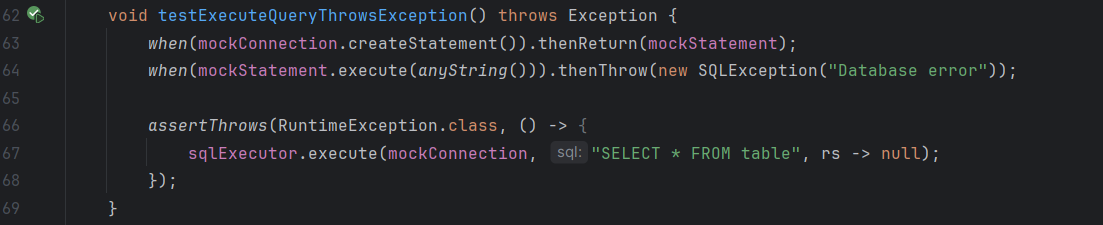
* **Objective**: Verify that the **SQLExecutor.execute()** correctly executes a SQL query and processes the **ResultSet** as expected.
* **Expected Result:** The test mocks a **ResultSet** containing a single row with the string "test result". It expects the **execute** method to return "test result" when it processes this **ResultSet**.



*The code of test case 1*

**Test Case 2: testExecuteQueryThrowsException**

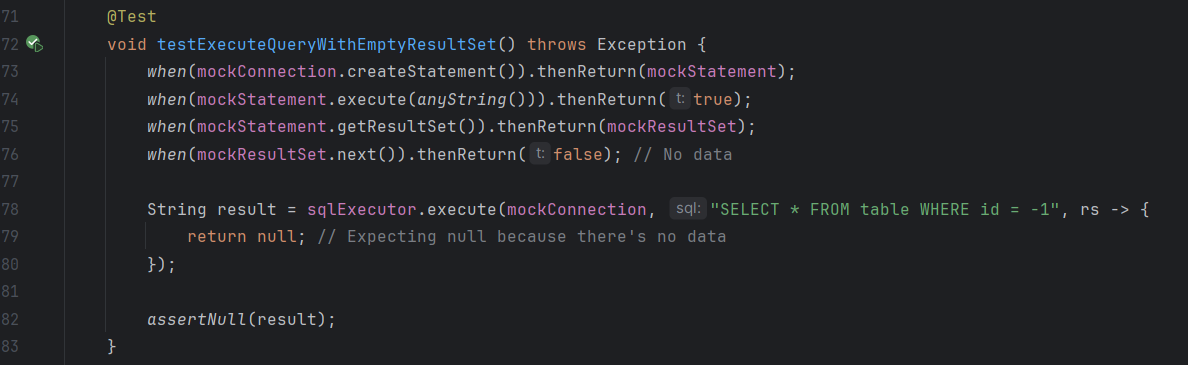
* **Objective**: This test checks the behavior of **SQLExecutor.execute()** when a **SQLException** is thrown during the execution of the SQL query.
* **Expected Result**: The test configures the mock Statement to throw a **SQLException** when execute is called. The test then expects that **SQLExecutor.execute()** will throw a **RuntimeException** in response to this **SQLException**.



*The code of test case 2*

**Test Case 3: testExecuteQueryWithEmptyResultSet**

* **Objective**: This test is designed to verify the behavior of **SQLExecutor.execute()** when the executed SQL query returns an empty **ResultSet**.
* **Expected Result**: The **ResultSet** is mocked to return **false** for its **next()** method, indicating that there are no rows in the **ResultSet**. The test expects that the **execute** method will return **null**, as there is no data to process from the **ResultSet**.



*The code of test case 3*

### Test “Press Save Button” JUnit Test Cases

The Press Button will save the console.

**Test Case 1: testExecuteQuery**

* **Objective**: Verify that the **OperationSavedController.create()** correctly saves the query console, and returns the expected result.
* **Expected Result:** The test expects the create method in **OperationSavedController** to return a **DataResult<Long>** object containing the expected data (in this case, 123L).



*The code of test case 1*

### Test “Press Format Button” JUnit Test Cases

The Format Button will format the SQL statement. If the SQL query is a one-line sentence, it will be formatted into multiple lines

**Test Case 1: testSqlFormat**

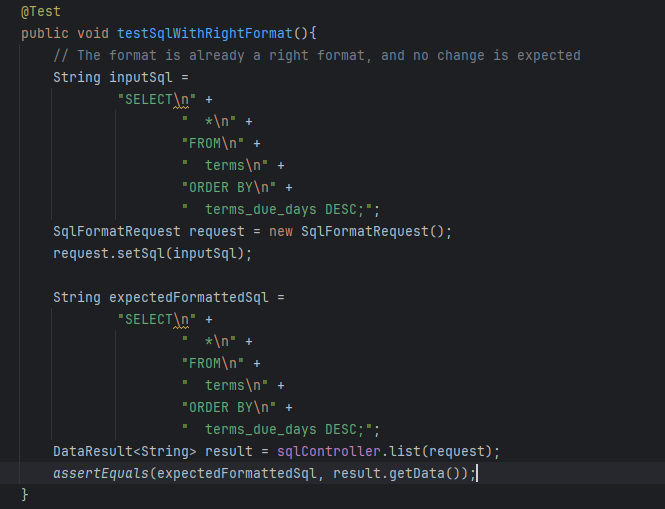
* **Objective**: Verify that the **SQLExecutor.execute()** correctly executes a SQL query and processes the **ResultSet** as expected.
* **Expected Result:** The test mocks a **ResultSet** containing a single row with the string "test result". It expects the **execute** method to return "test result" when it processes this **ResultSet**.



*The code of test case 1*

**Test Case 2: testSqlWithRightFormat**

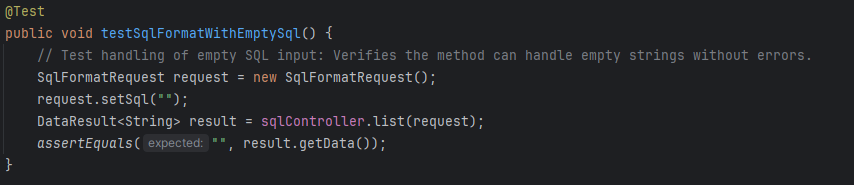
* **Objective**: To verify if a standard SQL statement is correctly formatted into a more readable multi-line format.
* **Expected Result:** The input SQL statement "**SELECT \* FROM terms ORDER BY terms\_due\_days DESC;**" should be formatted into a more readable multi-line format. The test checks if the actual formatted SQL matches the expected formatted SQL string.



*The code of test case 2*

**Test Case 3: testSqlFormatWithEmptySql**

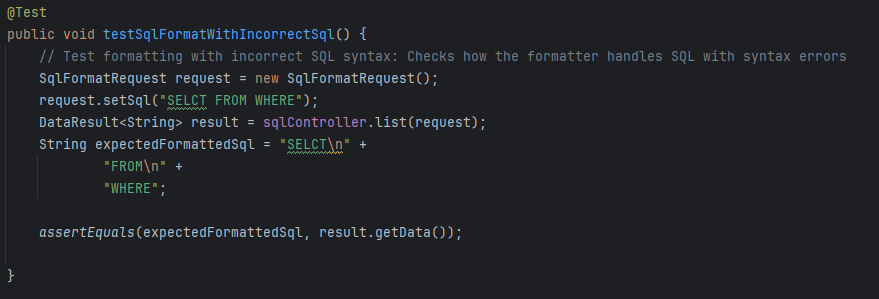
* **Objective**: To test the controller's behavior when provided with an SQL statement that is already in the correct format.
* **Expected Result:** Since the input SQL is already formatted, the test expects no change in the formatting. The output should match the input exactly.



*The code of test case 3*

**Test Case 4: testSqlFormatWithIncorrectSql**

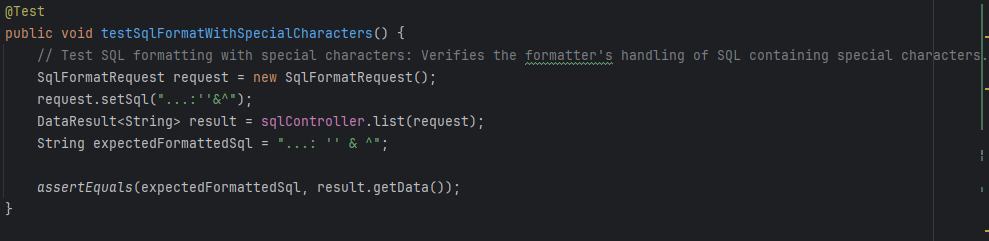
* **Objective**: To check how the formatter handles SQL statements with syntax errors.
* **Expected Result:** The test provides an incorrectly structured SQL string "**SELCT FROM WHERE**" and expects it to be formatted into a multi-line format, despite its syntax errors. It verifies whether the output matches the expected formatted SQL.



*The code of test case 4*

**Test Case 5: testSqlFormatWithSpecialCharacters**

* **Objective**: To verify the formatter's handling of SQL containing special characters.
* **Expected Result:** The test checks if the SQL formatter can correctly format an SQL string that contains special characters. The expected output is a formatted version of the input string, which contains various special characters.



*The code of test case 5*

# Part 3: White Box Testing and Coverage

## Structural Testing

Structural testing, also known as white-box testing, is a method of testing where the tester has knowledge of the internal structure, design, and implementation of the software. Unlike functional testing (also known as black-box testing), where testing is based on the specifications, structural testing focuses on how the system works internally.

Structural testing is important because it provides a more comprehensive evaluation of the software, as it ensures that all parts of the code have been tested. If part of the program is not executed by any test case in that suite, faults in that part cannot be exposed.

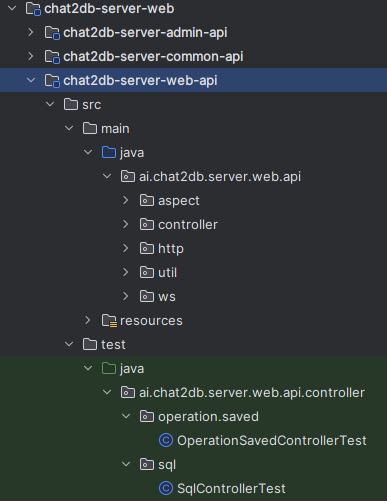
Structural also complements functional testing. It helps to detect errors and issues in the code, which might be missed by black-box testing. To include cases that may not be identified from specifications alone, we use the strategy of control flow testing to ensure that the various paths through a program's control structures are executed and tested.

## Coverage of Existing Test Suite

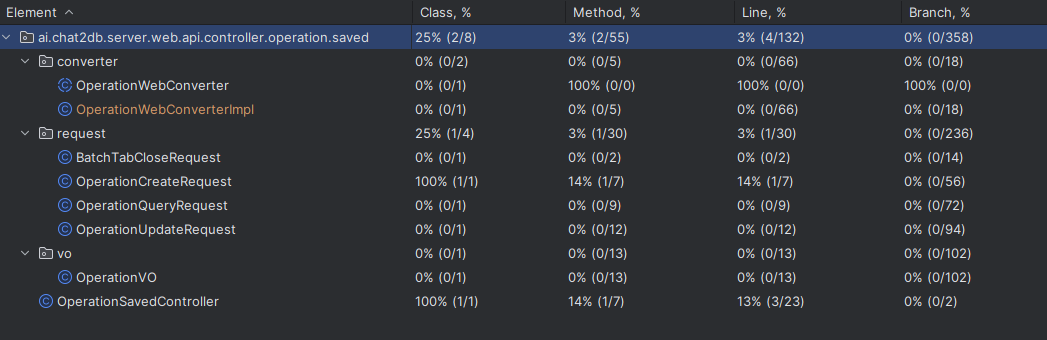
The coverage tool, JaCoCo (short for Java Code Coverage), is used for the test suite of this project. JaCoCo describes the degree to which the source code of a program is executed when a particular test suite runs. In order to use JaCoCo and test with Coverage, go to **Run > Edit Configurations…**, choose the **Junit** Application and go to **Modify options**. Choose **Specify alternative coverage runner option** and **Enable branch coverage and test tracking**. In the **Code Coverage option**, select **JaCoCo** from the **Choose coverage runner** dropdown.

Results of various coverage methods are shown for the existing test suite in chat2db-server-web-api, including:

1. Class, %
2. Method, %
3. Line, %
4. Branch, %



Existing Test Suite



Coverage Report

The coverage report shows that some parts of the code are currently uncovered.