**Project: Part 1**

**Introduction. Set Up. Functional Testing and Partitioning**

**2024-02-06**

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

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

# 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+js+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

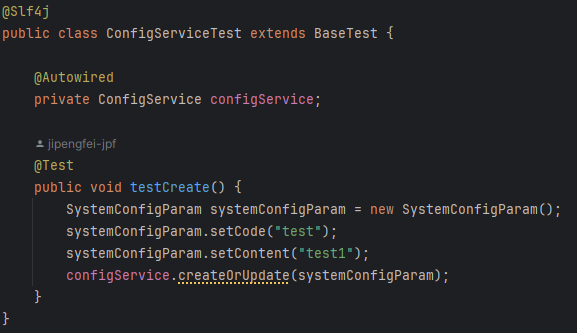
# Testcases:

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) and an invalid port designation ("x").
* **Null Port:** The port information is omitted, 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 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 successful connection using the default port.



*The code of test case 4*

# 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.