**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: Introduction. Set Up. Functional 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

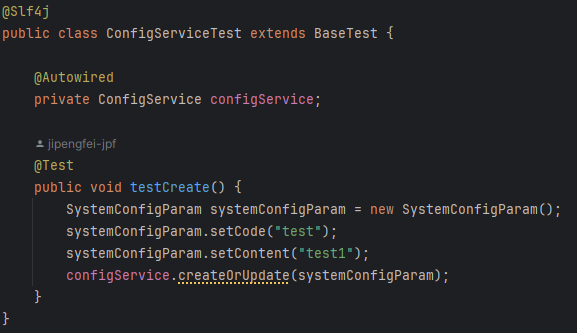
### Original Testcases

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)**

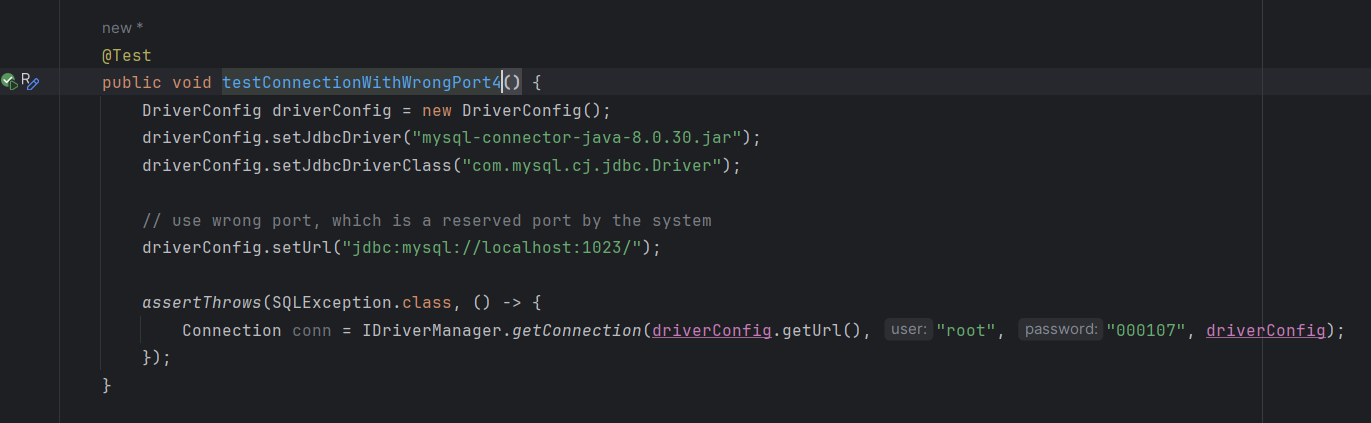
* **Objective**: Test the software's response to an invalid port designation (65536).
* **Method**: Configure **DriverConfig** with an invalid port representation.
* **Expected Result**: The test should throw an **SQLException**.



*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)**

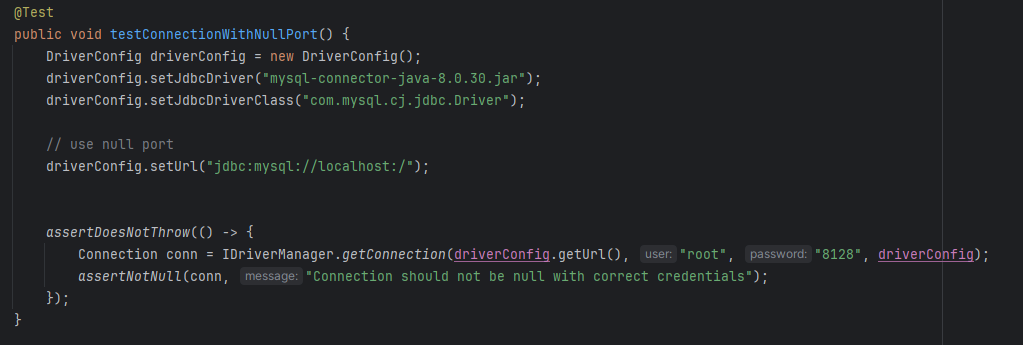
* **Objective**: Test the software's response to an invalid port designation (-1).
* **Method**: Omit the port information in the **DriverConfig**.
* **Expected Result**: The test should pass, indicating a successful connection using the default port.



*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:

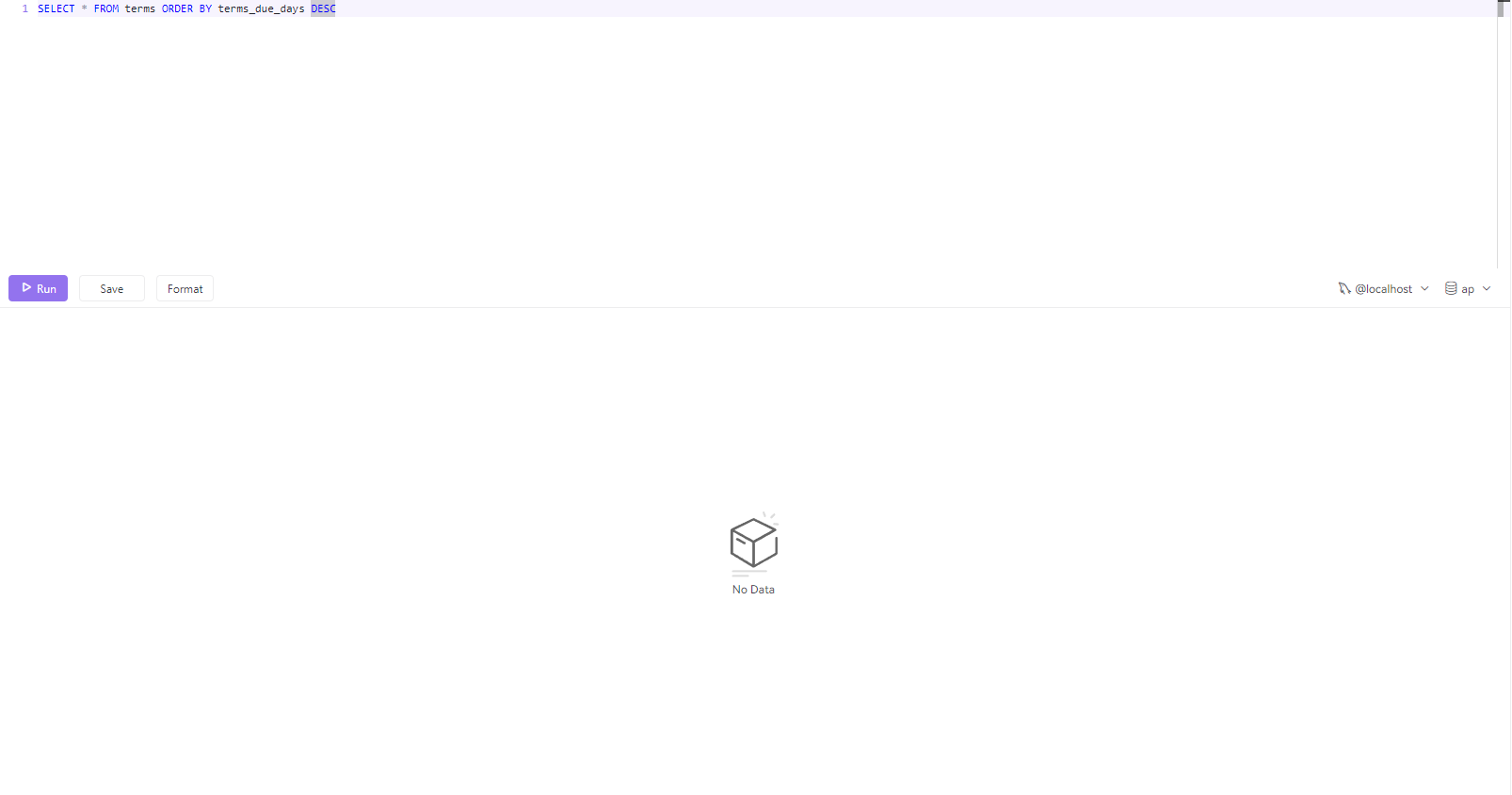
1. **Initial State**: This is the state when the SQL query is inputted as a single line.
2. **Formatted State**: After the "format" button is clicked, the SQL query is formatted into multiple lines.
3. **Saved State**: If the "save" button is clicked, the current console of the query is saved.
4. **Executed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

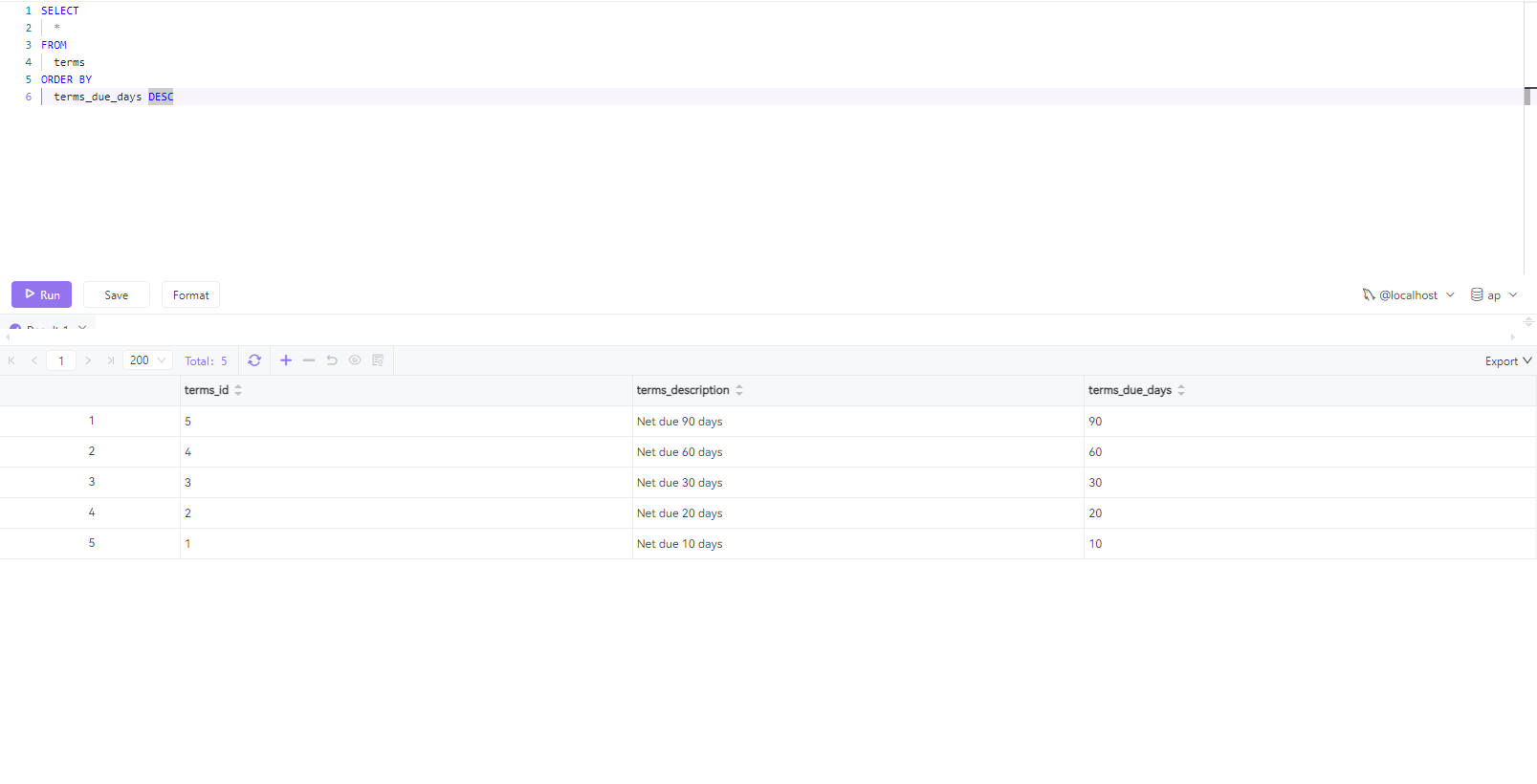
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| States | Description | Input: Press Button Run | Input: Press Button Save | Input: Press Button Format |
| Initial State | The SQL query is inputted as a single-line | **Executed State** | **Saved State** | **Formatted State** |
| Formatted State | The 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

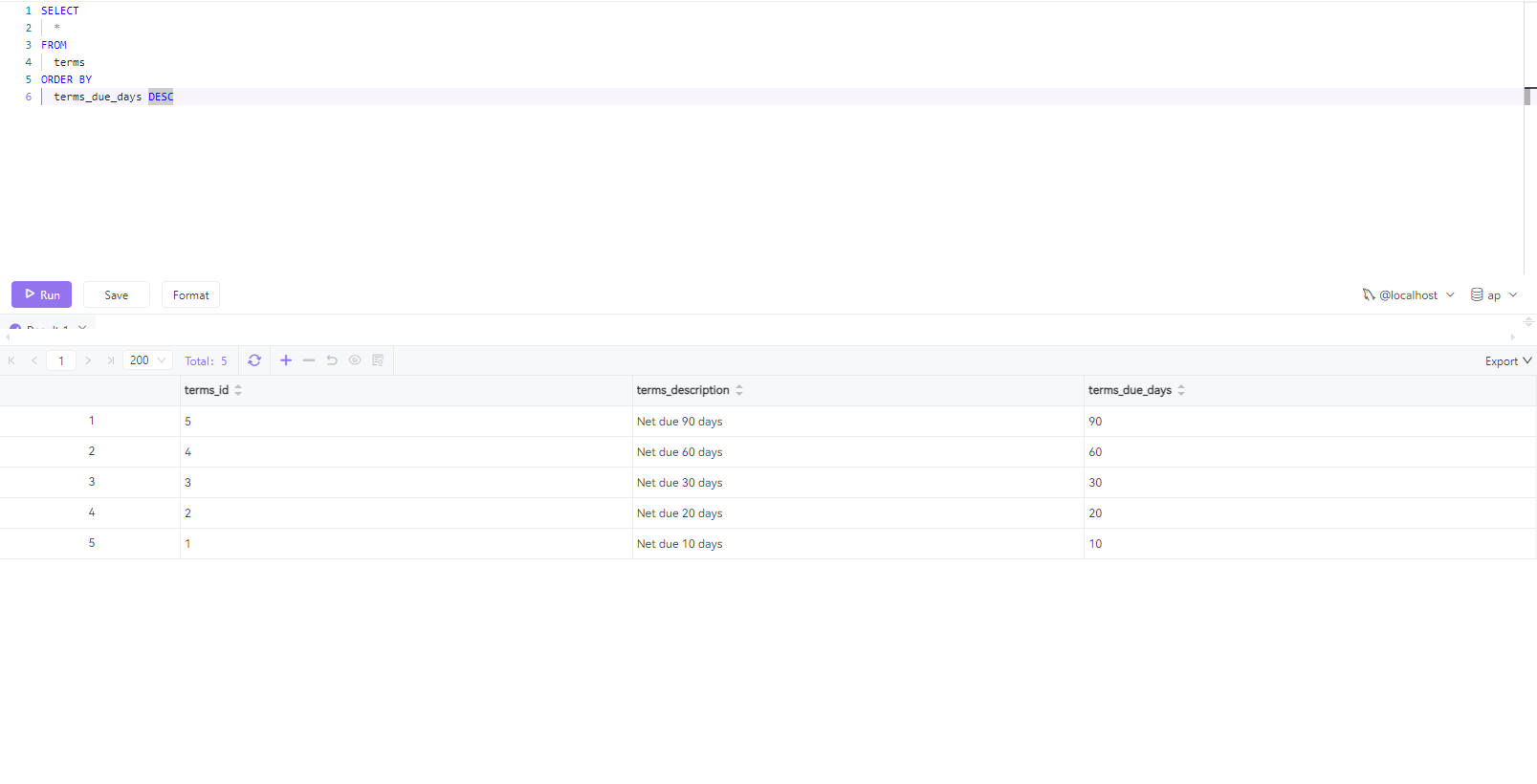
Some Screenshots for each of the states:



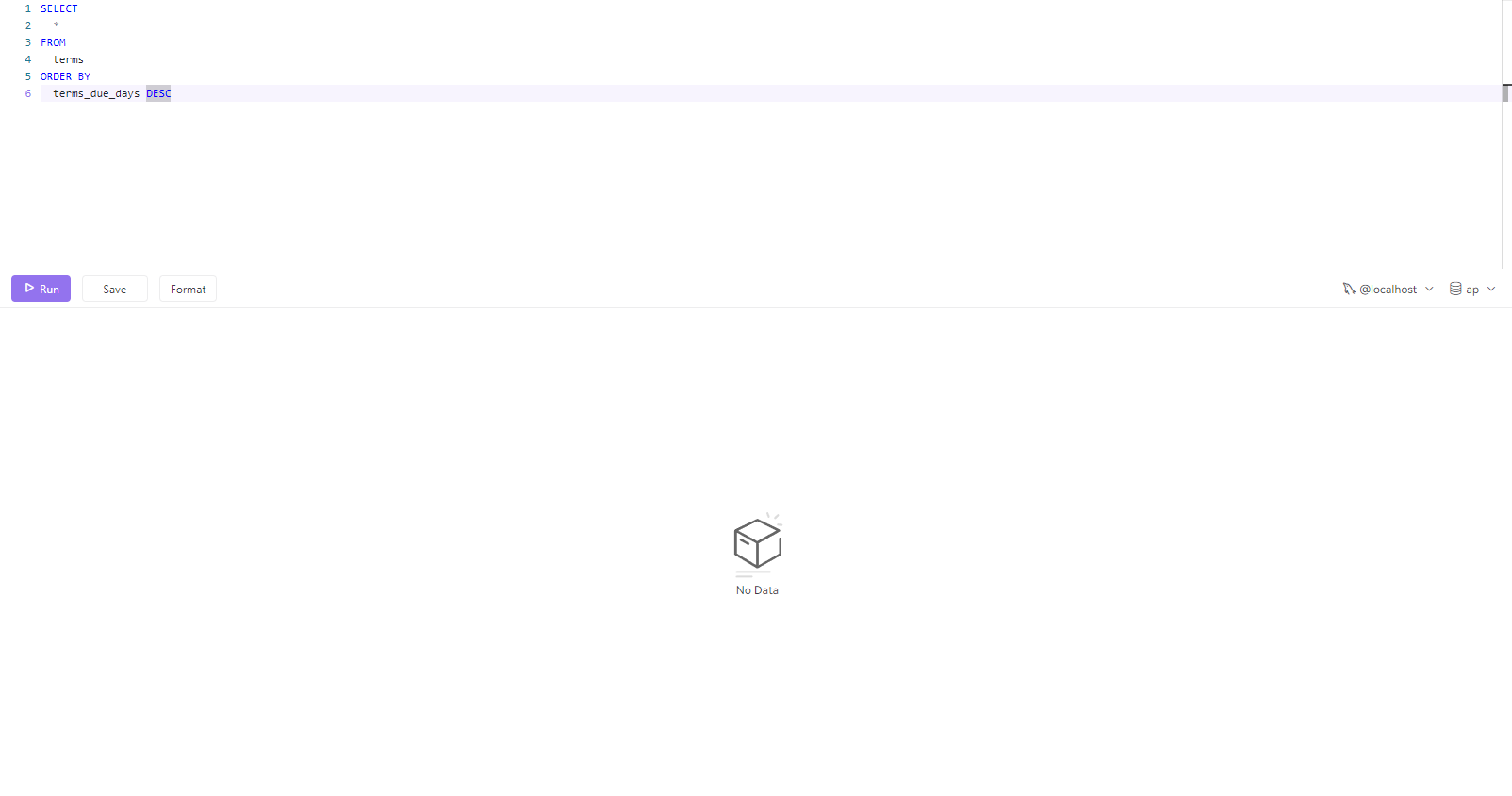
Initial State



Executed State

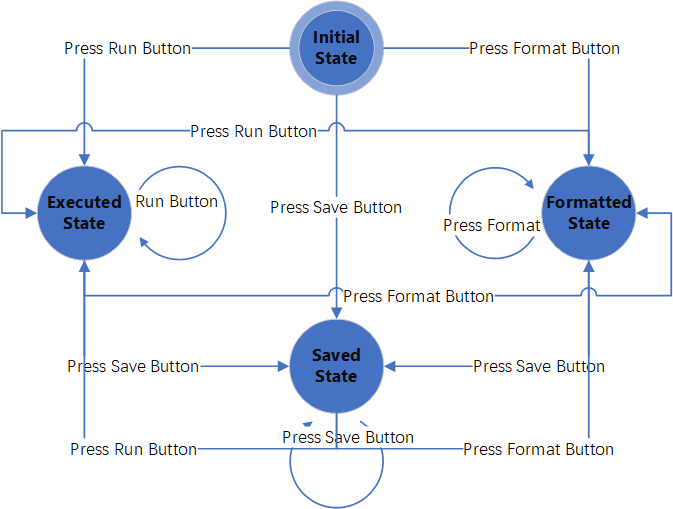


Saved State



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**

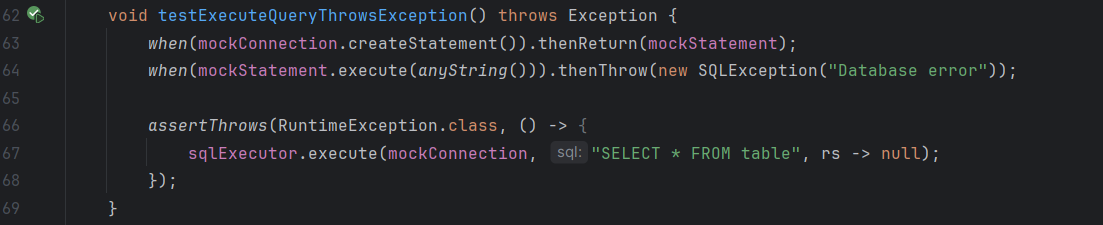
* **Objective**: Verify that the **SQLExecutor.execute()** correctly executes a SQL query and processes the **ResultSet** as expected.
* **Expected Result:** The test mocks a **ResultSet** containing a single row with the string "test result". It expects the **execute** method to return "test result"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



*The code of test case 1*

**Test Case 2: testExecuteQueryThrowsException**

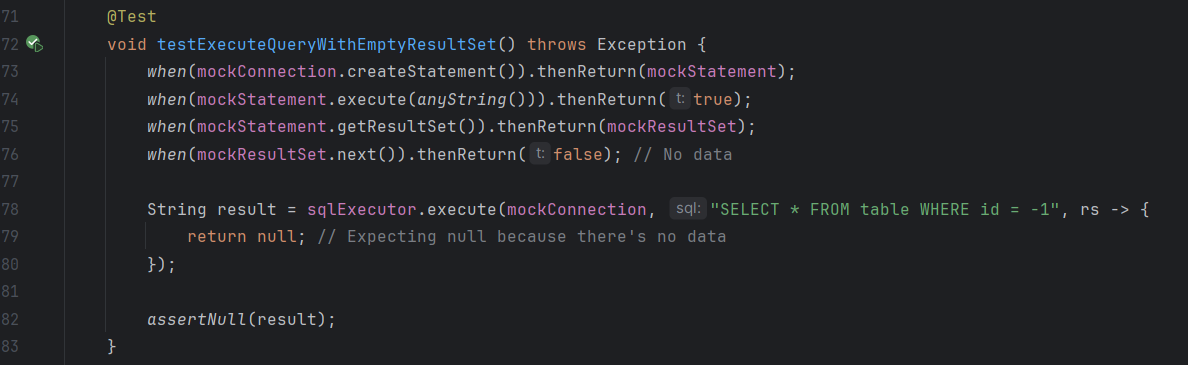
* **Objective**: This test checks the behavior of **SQLExecutor.execute()** when a **SQLException** is thrown during the execution of the SQL query.
* **Expected Result**: The test configures the mock Statement to throw a **SQLException** when execute is called. The test then expects that **SQLExecutor.execute()**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

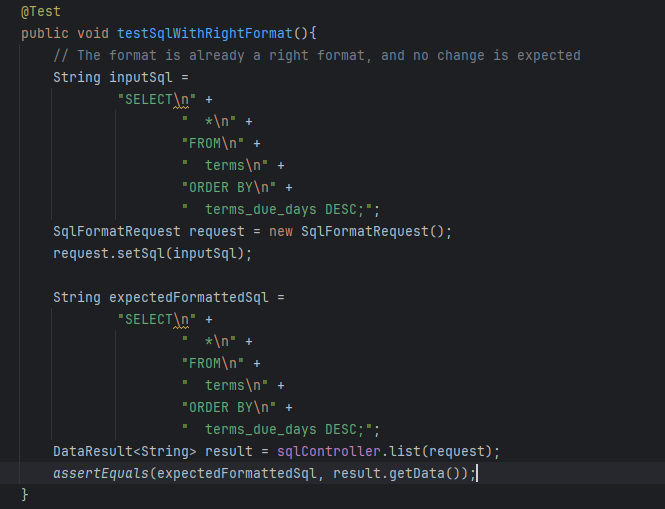
* **Objective**: Verify that the **SQLExecutor.execute()** correctly executes a SQL query and processes the **ResultSet** as expected.
* **Expected Result:** The test mocks a **ResultSet** containing a single row with the string "test result". It expects the **execute** method to return "test result"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



*The code of test case 1*

**Test Case 2: testSqlWithRightFormat**

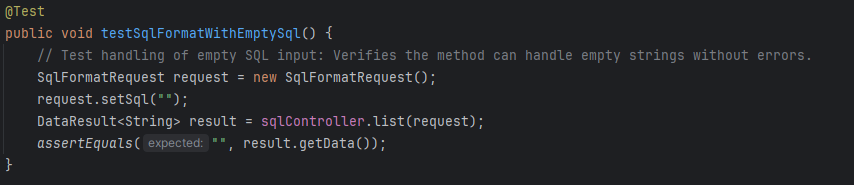
* **Objective**: To verify if a standard SQL statement is correctly formatted into a more readable multi-line format.
* **Expected Result:** The input SQL statement "**SELECT \* FROM terms ORDER BY terms\_due\_days DESC;**" should be formatted into a more readable multi-line format. The test checks if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



*The code of test case 2*

**Test Case 3: testSqlFormatWithEmptySql**

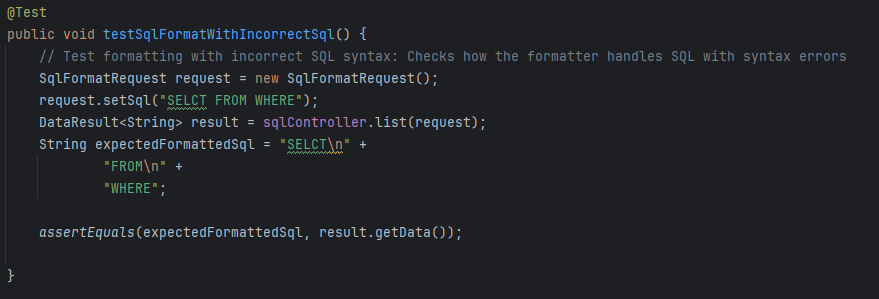
* **Objective**: To test the controller's behavior when provided with an SQL statement that is already in the correct format.
* **Expected Result:** Since the input SQL is already formatted, the test expects no change in the formatting. The output should match the input exactly.



*The code of test case 3*

**Test Case 4: testSqlFormatWithIncorrectSql**

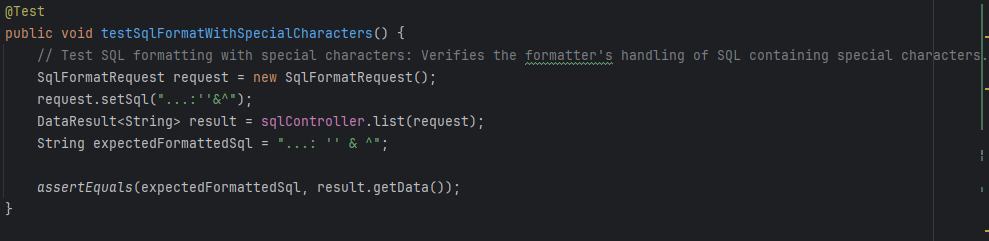
* **Objective**: To check how the formatter handles SQL statements with syntax errors.
* **Expected Result:** The test provides an incorrectly structured SQL string "**SELCT FROM WHERE**" and expects it to be formatted into a multi-line format, despite its syntax errors. It verifies whether the output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

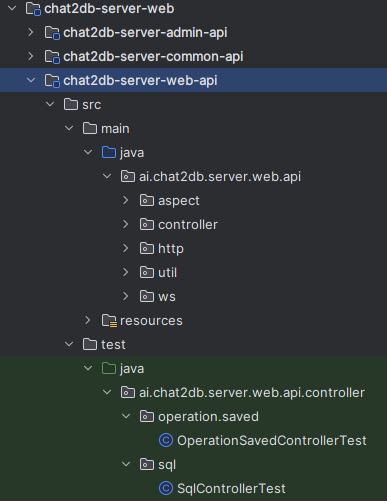
Structural also complements functional testing. It helps to detect errors and issues in the code, which might be missed by black-box testing. To include cases that may not be identified from specifications alone, we use the strategy of control flow testing to ensure that the various paths through a program's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

## Coverage of Existing Test Suite

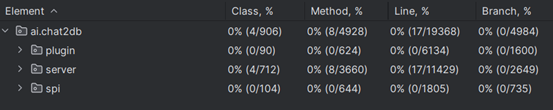
The coverage tool, JaCoCo (short for Java Code Coverage), is used for the test suite of this project. JaCoCo describes the degree to which the source code of a program is executed when a particular test suite runs. In order to use JaCoCo and test with Coverage, go to **Run > Edit Configurations…**, choose 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.

## Code uncovered by the Existing Test Suite

**Test suite: Chat2DBAzureAIIntegrationTests**

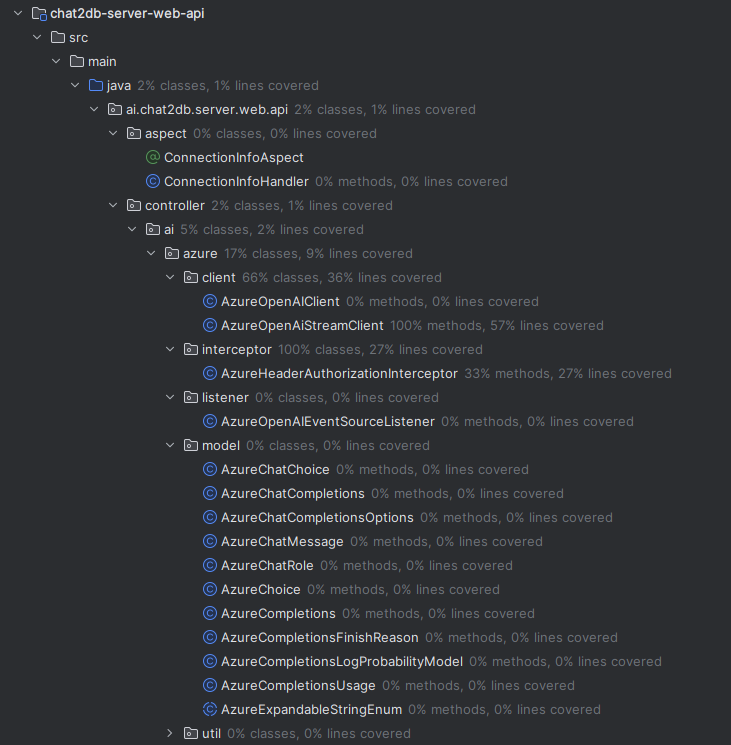


Figure of uncovered code

Chat2DB, which is an intelligent and versatile general-purpose SQL client and reporting tool for databases, the untested code under "chat2db-server-web-api" are particularly significant. This is because they are integral to the AI interactions within the Chat2DB system, which handles the natural language understanding and processing for database operations.

* **AI**: The "ai" package within the Chat2DB server's web API module is a comprehensive suite dedicated to integrating various artificial intelligence services into the Chat2DB system. This package is pivotal as it encapsulates all the necessary components that allow Chat2DB to interact with external AI providers, harnessing their capabilities to process and understand natural language queries for databases.
* **AI Controllers (e.g., AiConfigController, ChatController)**: These controllers are likely responsible for handling API requests related to AI operations, such as configuring AI settings, initiating chat operations, and managing embeddings or knowledge bases. Testing these controllers is crucial for verifying that the Chat2DB API endpoints function correctly and securely, providing a reliable interface for the AI operations.
* **Utility Classes (e.g., enums, config, converter)**: Although not directly related to AI, these classes support the AI components by providing necessary configurations, enumerations, and data conversion utilities. They ensure that the AI components can operate with the correct parameters and data formats, which is essential for the accurate functioning of the AI features.
* **Operation and Task Sections**: These sections may handle the orchestration of AI tasks and the operational logic behind processing database queries. They are essential for ensuring that the AI understands the user's intent and can translate it into actionable database commands.
* **System and User Sections**: These are likely involved in managing system-wide configurations and user-related data, which might influence how the AI components interact with each user or system setting. The proper functioning of these sections is important for personalizing the AI experience and maintaining system stability.
* **Response and Request Sections**: Contain the classes for handling the request and response objects within the AI components. They are important for ensuring that communication between the server and AI services is correctly structured and that the data exchanged is properly encapsulated.

The AI capabilities are at the heart of Chat2DB's value proposition—allowing users to interact with databases through natural language. Without thorough testing of these components, the core functionality of Chat2DB might be compromised, leading to inaccuracies in database querying, poor user experience, and potential security vulnerabilities.

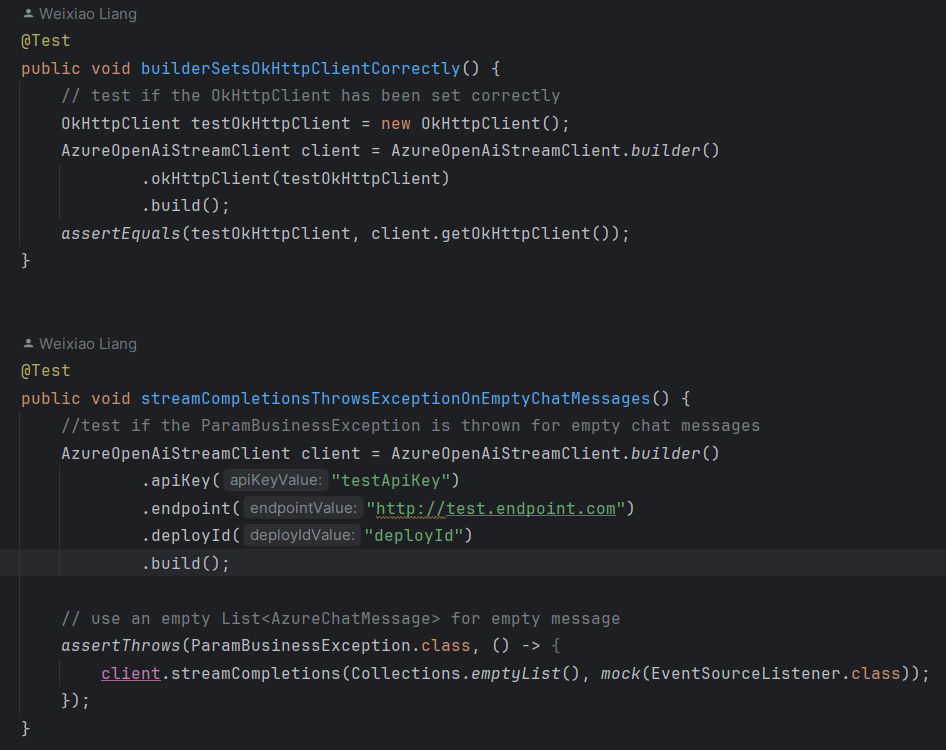
In summary, the untested AI-related code within the Chat2DB server's web API is a foundational element that enables the intelligent interpretation and handling of database queries. Ensuring these components are fully tested guarantees that Chat2DB can deliver on its promise to provide an intuitive, reliable, and secure AI-driven database interaction experience.

## New Test Cases

The newly added test cases cover critical functionalities of the **AzureOpenAiStreamClient** and **BaichuanAIStreamClient** within the Chat2DB server project. These clients are designed for integrating Chat2DB with external AI services, enabling the application to leverage AI for processing and understanding natural language queries against databases. The test cases aim to ensure the reliability, configurability, and robust error handling of these integrations.

### AzureOpenAiStreamClient Test Cases

1. **Builder Pattern Verification**: Tests verify that the builder pattern implemented for **AzureOpenAiStreamClient** correctly sets up the API key, endpoint, deploy ID, and custom **OkHttpClient**. These properties are essential for establishing a secure and customized connection to Azure's AI services, ensuring that the client can successfully authenticate and communicate with Azure.
2. **Exception Handling for Empty Chat Messages**: A test case checks the client's behavior when provided with an empty list of chat messages for the **streamCompletions** method. This method is crucial for sending user inputs to Azure AI and receiving AI-generated responses. The test ensures that the client robustly handles invalid inputs by throwing a **ParamBusinessException**, preventing the application from making unnecessary calls to Azure AI with empty data.



AzureOpenAiStreamClient Test Cases

### BaichuanAIStreamClient Test Cases

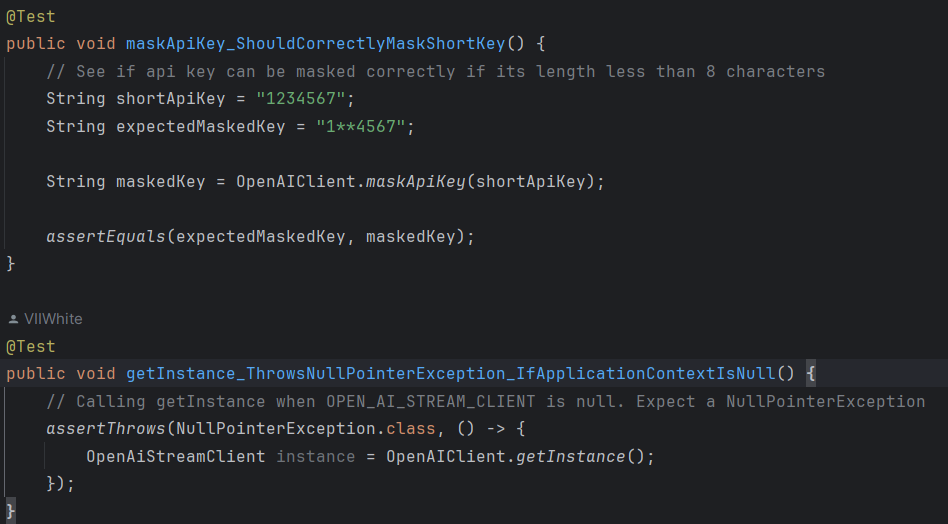
1. **Builder Pattern Verification**: Similar to the Azure client, these tests confirm that the builder pattern for **BaichuanAIStreamClient** accurately sets API keys, secret keys, API hosts, models, and embedding models. This setup is crucial for integrating with **Baichuan's** AI services, allowing for a wide range of configurations to match the specific AI capabilities and security requirements needed by Chat2DB.
2. **Handling of Empty Chat Messages**: The test for **streamCompletions** method functionality with empty chat messages ensures that the **Baichuan** client, like its Azure counterpart, properly handles error scenarios. It verifies that the system is safeguarded against making calls with invalid data, which is important for maintaining the efficiency and reliability of the AI integration.



BaichuanAIStreamClient Test Cases

### OpenAIClient Test Cases

1. **Masking API Key:** These tests validate the behavior of the **maskApiKey** method in the **OpenAIClient** class. They verify that the method correctly masks the API key by replacing a portion of it with asterisks. The tests cover scenarios such as masking a valid key, masking a null key, and masking a short key, ensuring comprehensive coverage of the masking functionality.
2. **Null Pointer Exception Handling:** This test ensures that the **getInstance** method in the **OpenAIClient** class behaves correctly when provided with a null **ApplicationContext**. It checks whether the method reacts as expected by throwing an error, which helps prevent potential issues during runtime and ensures the reliability of the client's instantiation process.



OpenAIClient Test Cases

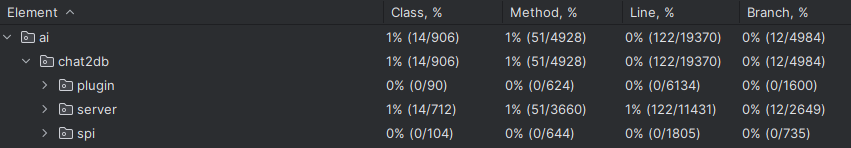
### Importance

These test cases are vital for several reasons:

* **Ensure Correct Configuration**: They confirm that both clients can be correctly configured through their builders, which is essential for customizing the integration based on the deployment environment and security policies.
* **Validate Error Handling**: By verifying how the clients handle empty inputs, the tests ensure that the application can gracefully manage error scenarios, improving the robustness and user experience of Chat2DB.
* **Facilitate Integration Testing**: These tests are a foundational step towards more comprehensive integration testing with Azure and Baichuan AI services, ensuring that the application can interact with external AI services as expected.
* **Improve Test Coverage**: By covering critical functionalities and error handling paths, these tests contribute to the overall test coverage of the Chat2DB project, helping to identify potential issues early in the development cycle and ensuring the reliability of the application.

In summary, the addition of these test cases to the testing document enhances the assurance of the Chat2DB server's AI integration components, ensuring they are correctly configured, secure, and capable of handling errors gracefully.

### Coverage after create new test cases



Coverage Report create new test cases

After adding new test cases, the total number of lines increased from the previous 17 lines to 122 lines, representing a growth of 105 lines.

# Part 4: Continuous Integration

## Continuous Integration

Continuous integration (CI) is a software development practice where developers regularly merge their code into a central repository, and the project is then automatically built and tested. Compared with integration, CI requires developers to commit their code frequently, even with small changes. Also, CI is an automated process, each integration is verified by an automated build and automated tests to detect integration errors as quickly as possible.

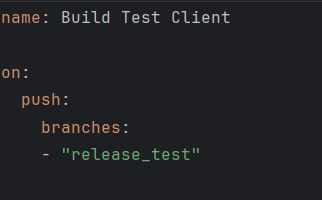
## CI Systems

Continuous Integration systems are tools that automate the process of integrating code changes from multiple contributors into a single software project. They run tests and builds automatically, ensuring that new code contributions do not break the existing codebase. Some popular CI systems include GitHub Actions, Jenkins, Travis CI, CircleCI, and so on. Most modern CI systems require a YAML file to define the configuration for the CI/CD pipeline.

## GitHub Actions

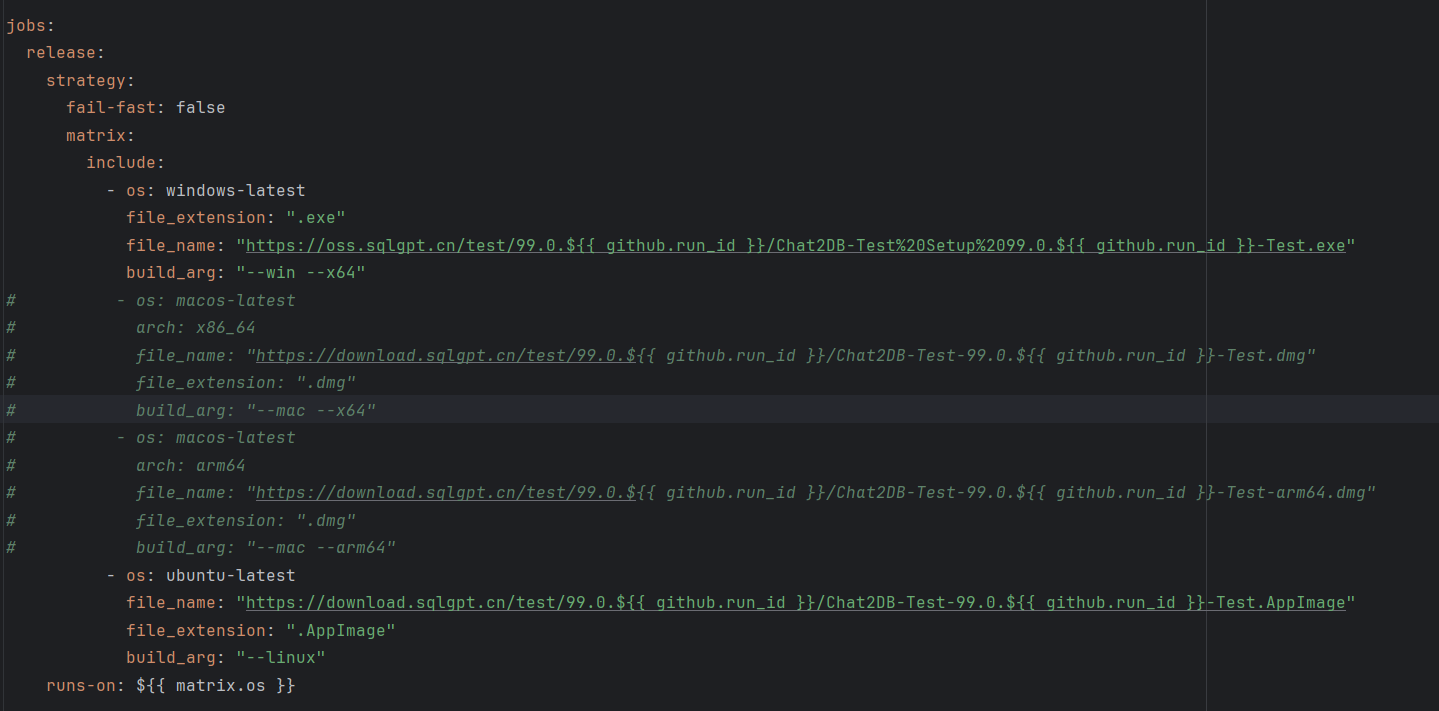
Chat2DB uses GitHub Actions for CI/CD service. By creating a YAML configuration file in path .github/workflows, and specifying the event triggers, the build, test, and deployment tasks can be directly finished within the GitHub repository.

For example, in the file “release\_test.yml”, event triggers are specified as bellows:



Specify Event Triggers

This suggests when the changes are pushed to the branch “release\_test”, the job will be executed automatically on GitHub. Also, the job will run on various operating systems such as Windows and Ubuntu:



Specify Operating Systems

The job includes multiple steps:

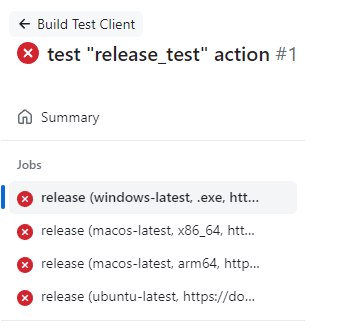
1. Check out Repository: Checks out the code in the git repository to the runner. It allows the workflow to get access to the code in the repository.
2. Install JRE: Installs Java Runtime Environment (JRE) version 17 and “temurin” distribution.
3. Enable TLS 1.0 and 1.1 in java.security: Modifies the java.security file to enable TLS 1.0 and 1.1.
4. Copy JRE to Static Directory: Creates a directory named chat2db-client/static and copies the JRE to this location. Changes permissions if the OS is not Windows.
5. Delete File on Linux: If the runner is Linux, this step deletes the legal directory from the JRE in chat2db-client/static/jre/.
6. Install Node.js: Installs Node.js version 16 using the actions/setup-node action. It also caches dependencies based on the yarn.lock file.
7. Install Java and Maven: Installs Java and Maven for building Java applications.
8. Build FE Static: Executes commands to build the front-end static resources for the web and copy them to the server’s static and thymeleaf directories.
9. Build BE Static: Builds the backend project using Maven and copies the jar file and version to specific directories.
10. Prepare Build Electron: Prepares the build for Electron.
11. Build/Release Electron App: Uses the samuelmeuli/action-electron-builder action to build the Electron application.
12. Prepare Upload for Jar: Prepares for uploading jar file, including zipping and copying related files to a temporary directory.

## Practices

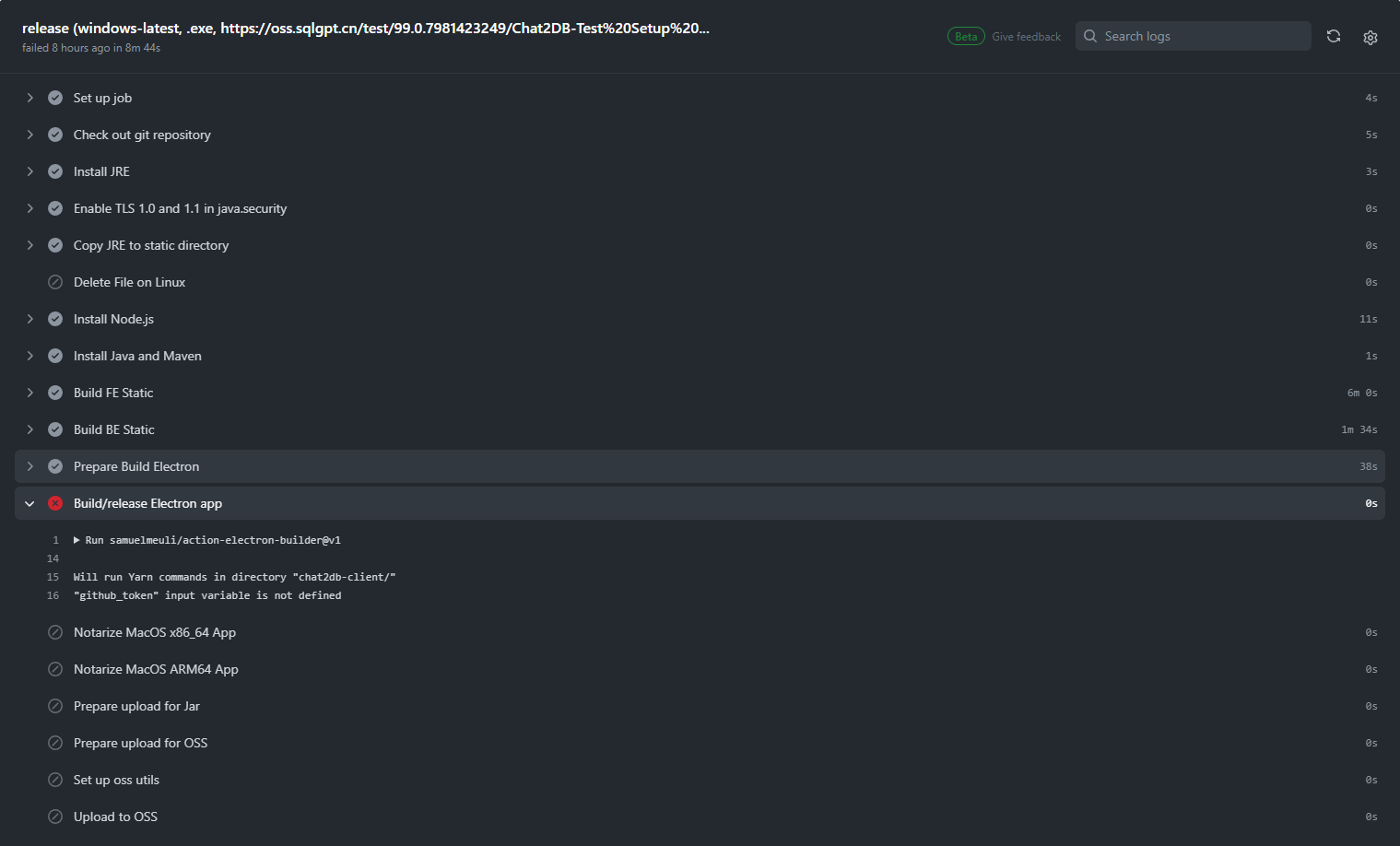
### test "release\_test" action #1

All four Jobs of the test "release\_test" action #1 failed different operating systems or architectures.

1. Windows-latest
2. macOS-latest, x86\_64
3. macOS-latest, arm64
4. Ubuntu-latest



Jobs List

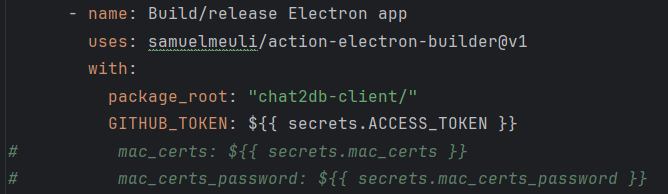


Failed Job on windows

From the error, we know it requires a “github\_token” variable. In order to add the token, a GitHub token of the repository owner is generated and added to the Secrets of Chat2DB repository, with the name of ACCESS\_TOKEN.

### test "release\_test" action #2

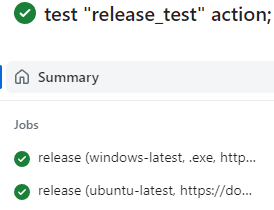
For the second practice, two operating systems of macOS are removed from the configuration file, since they require extra Secrets for certification used to distribute applications for macOS.



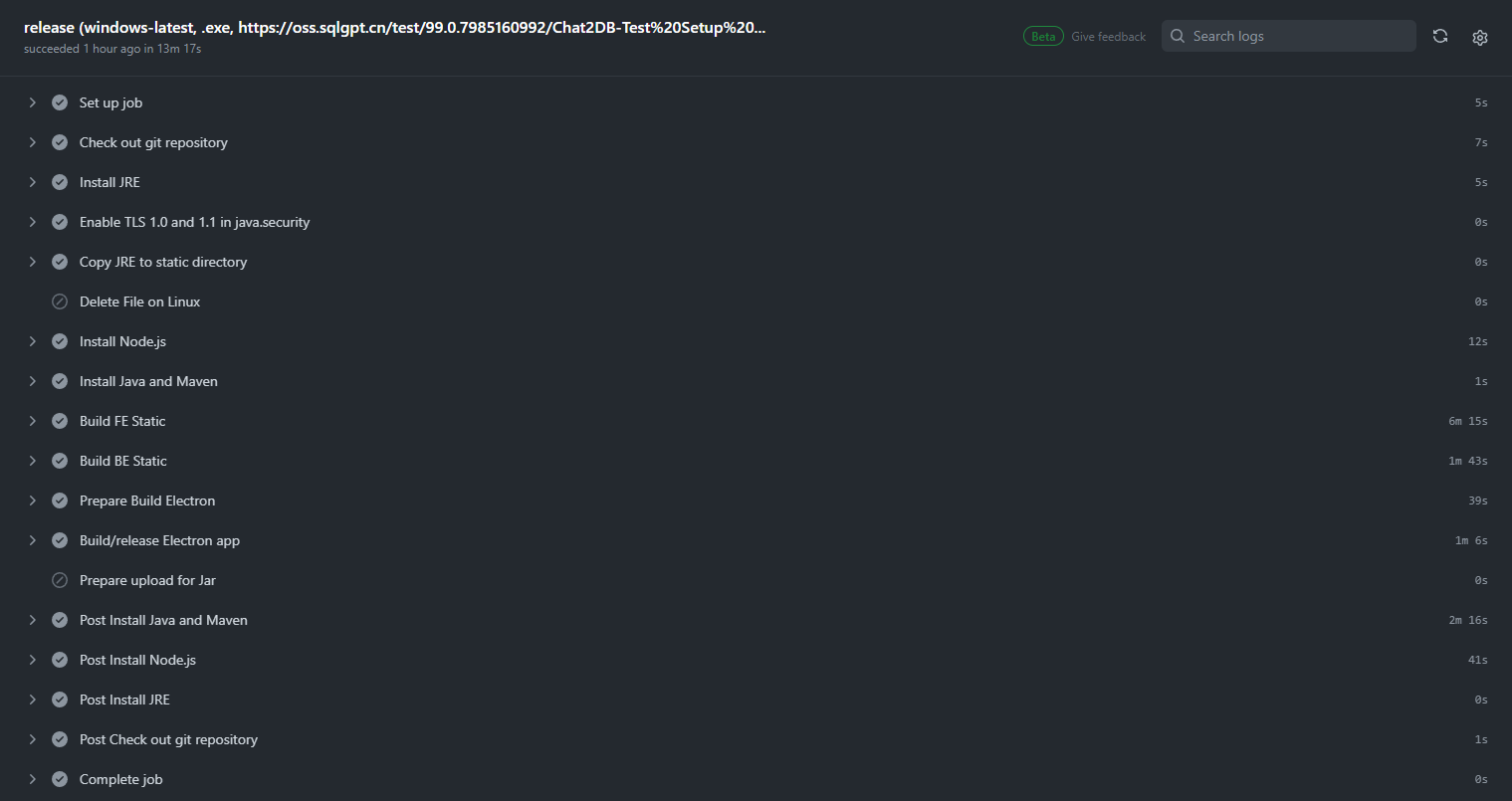
Remove Secrets Used by macOS

Some unnecessary steps such as uploading project file to OSS and server, as well as sending messages to office tool are also removed from the configuration file.

Then the changes are pushed to “release\_test” branch and all two jobs passed.



Jobs List



All Steps Passed

# Part 5: Testable Design & Mocking

## Testable Design

A testable design refers to the way a system or component is structured and implemented to make it easier and more efficient to test. A testable design ensures early bug detection, thus improves software quality. It also facilitates change and maintenance and improves development efficiency with its modular and loosely coupled structure.

There are some methods to improve testability of codes:

1. **Avoid Complex private Methods**: Private methods are inaccessible to tests outside the class, so complex logic within them can be a source of undetected bugs. Only use private methods for simple functions when necessary.
2. **Avoid Static Methods**: Static methods belong to the class, not instances of the class, and can't be easily overridden or mocked during testing. Use non-static instead, especially when there’s a need to use mock or stub.
3. **Be Careful Hardcoding new**: Hardcoding object creation with new makes it difficult to replace these objects with mocks or stubs in tests. Use dependency injection, where objects are created outside and passed in.
4. **Avoid Logic in Constructors**: Constructors can't be bypassed, and their logic will always execute, making it hard to isolate for testing. Move complex logic out of constructors and into methods.
5. **Avoid Singleton Pattern**: The Singleton Pattern, which restricts a class to a single instance, can make testing difficult because it introduces a global state into an application. Avoid or cautiously use Singleton patterns.

## Example of Less Testable Code

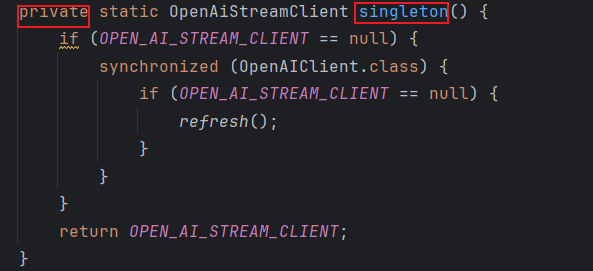
In **OpenAIClient** Class of path “chat2db-server-web-api/src/main/java/ai.chat2db.server.we.api/controller/ai/openai/client”, there are some implementations that make it less testable.

1. Frequent use of static methods:



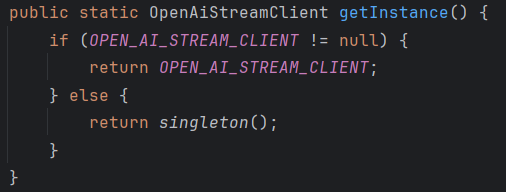
All methods in the class are static methods. They should be created as non-static methods to make the code more testable.

1. Use of private method **singleton()** and Singleton Pattern:



The **singleton()** method is a private method, and implements the Singleton Pattern. The OpenAiStreamClient instance could be created somewhere else.

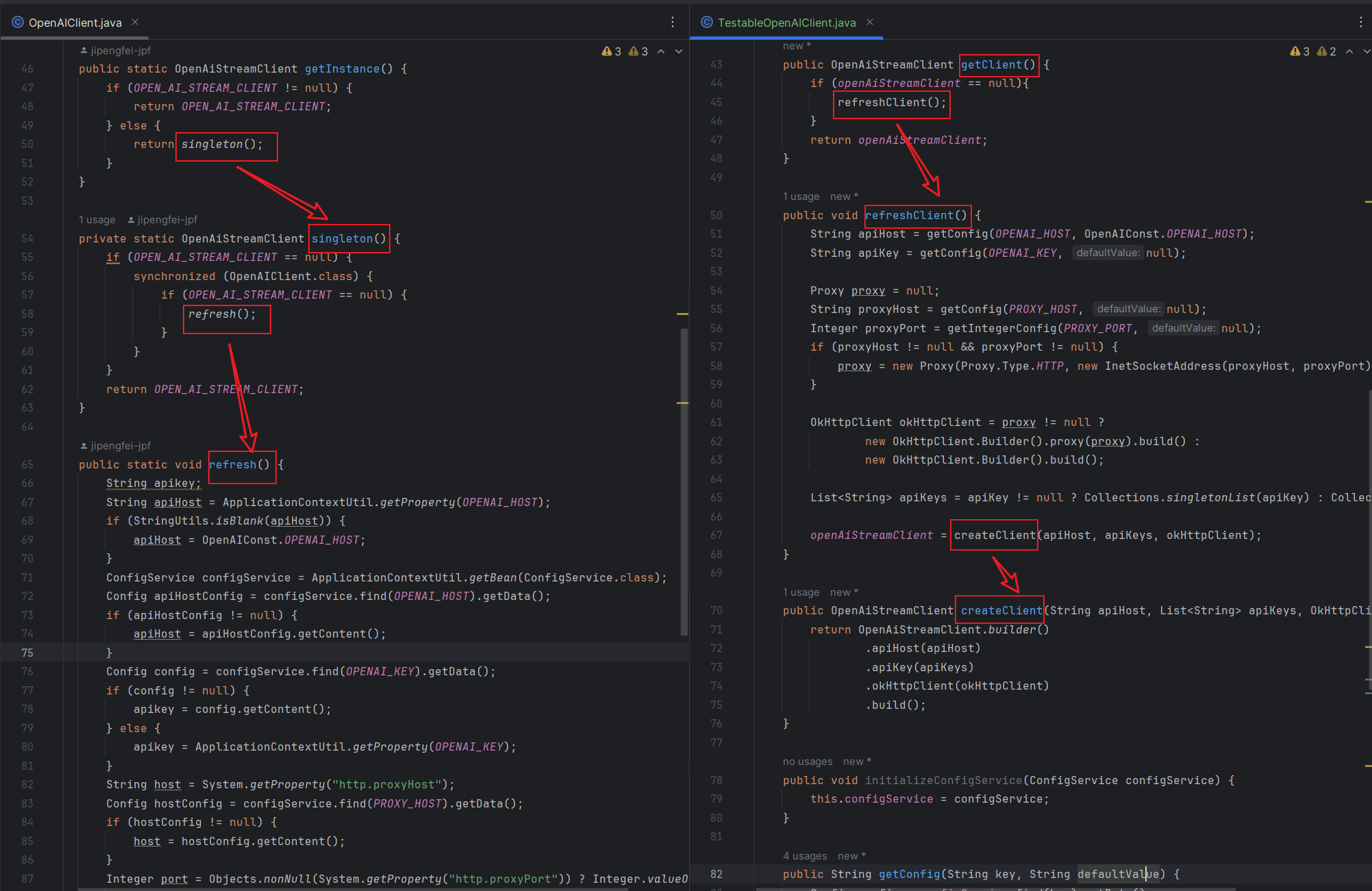
1. Use of static **getInstance()** method to get a instance of singleton object.



The **getInstance()** method is also part of the Single Pattern. It should be replace with dependency injection to improve testability.

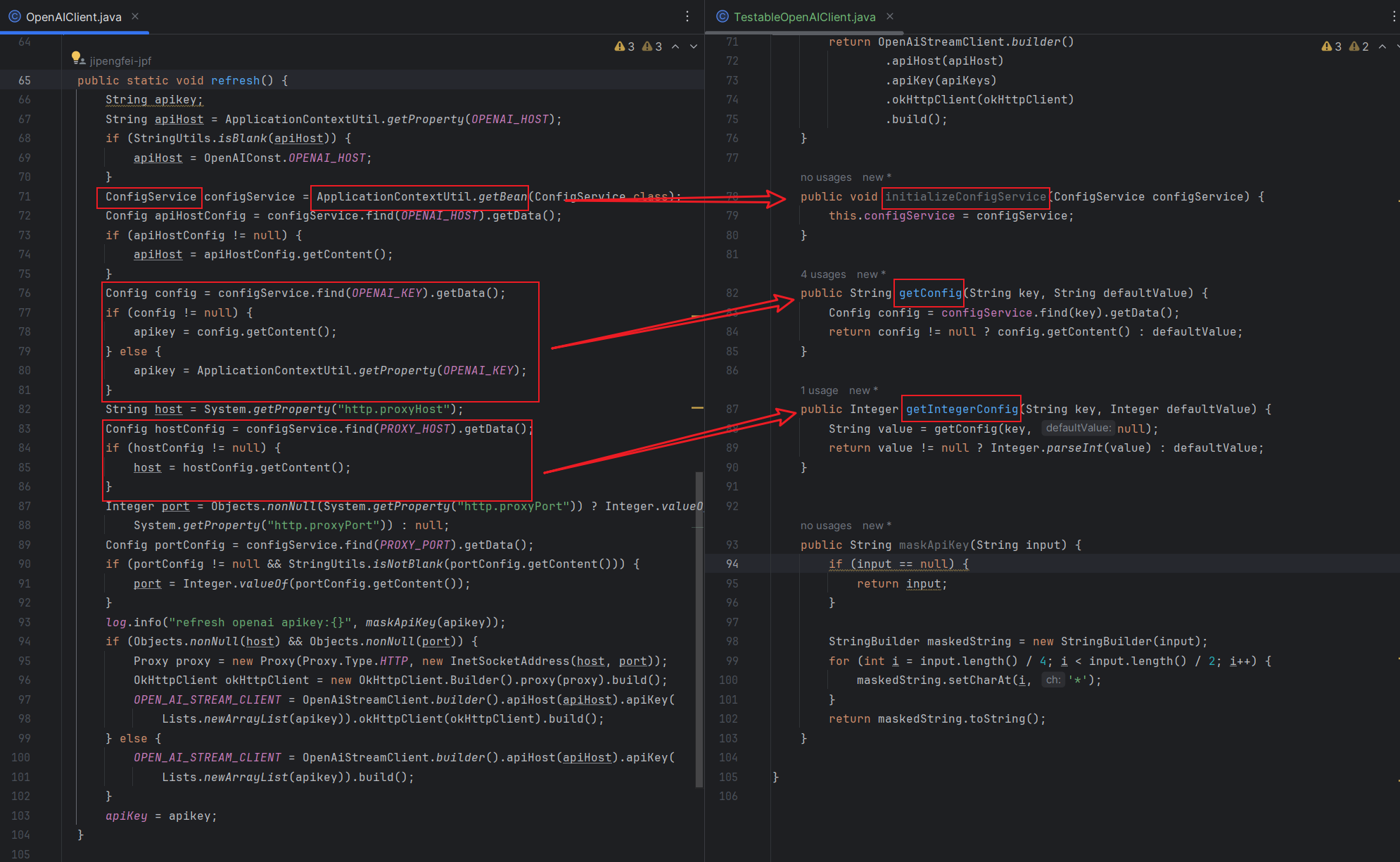
## Practice to Improve Testability

### Modification 1



The **OpenAiStreamClient** used to be created using static method **getInstance()** calling private static method **singleton()** with Singleton Pattern, and is now created using public non-static method **getClient()** calling public non-static method **refreshClient()** without Singleton Pattern.

### Modification 2



**ConfigService** used to be created in the public static method **refresh()**, but is now created using dependency injection in public non-static method **initializeConfigService()**. Codes to set apiKey and host value are also encapsulated and decoupled from original method.

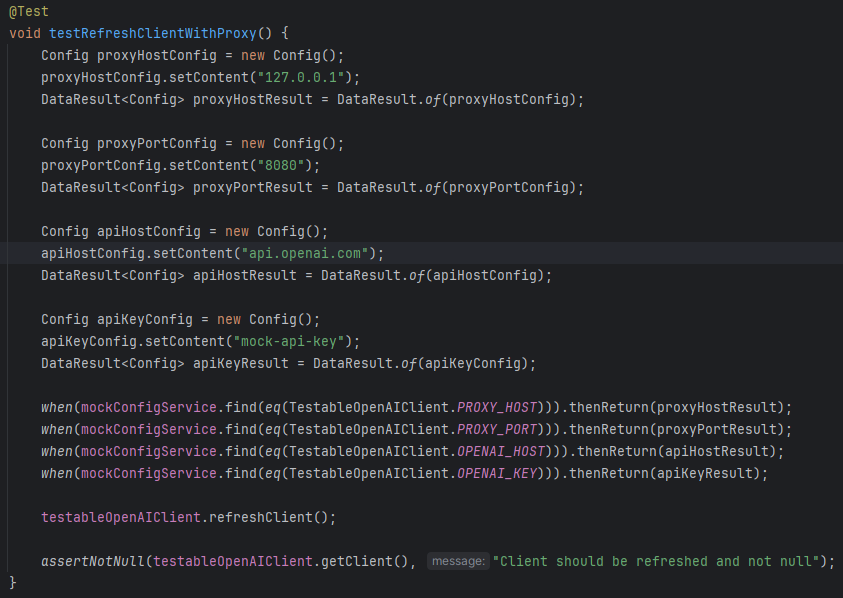
### Modification 3



Static method **maskApiKey()** is also converted into non-static method.

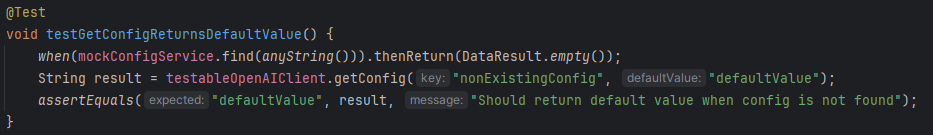
## Test Cases for Modified Testable Class

**Test Case 1: testRefreshClientWithProxy**



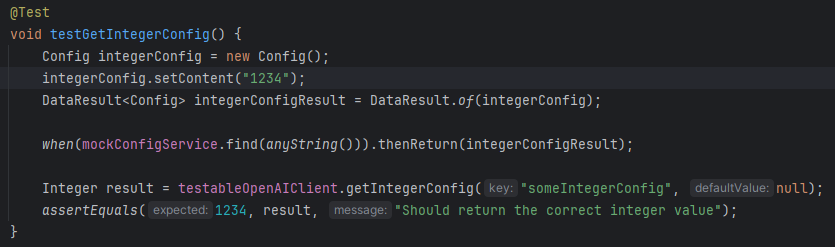
* **Objective:** To verify that the **TestableOpenAIClient** can refresh its configuration with the correct proxy settings.
* **Expected Result:** The test checks if the client is properly refreshed with the proxy host, proxy port, OpenAI host, and API key settings. The expected outcome is that the client is initialized and not null after the refresh.

**Test Case 2: testGetConfigReturnsDefaultValue**



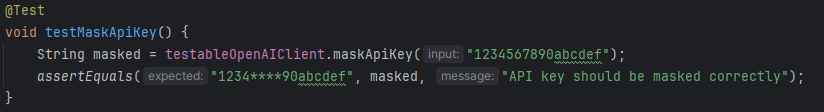
* **Objective:** To ensure that the **TestableOpenAIClient** returns a default value when a configuration setting is not found.
* **Expected Result:** The test checks if the **getConfig** method returns the specified default value when the requested configuration is not present. The expected output is the default value provided in the method call.

**Test Case 3: testGetIntegerConfig**



* **Objective:** To confirm that the **TestableOpenAIClient** correctly retrieves an integer value from the configuration.
* **Expected Result:** The test checks if the **getIntegerConfig** method correctly parses and returns an integer value from the configuration. The expected result is the integer value that was set in the configuration.

**Test Case 4: testMaskApiKey**

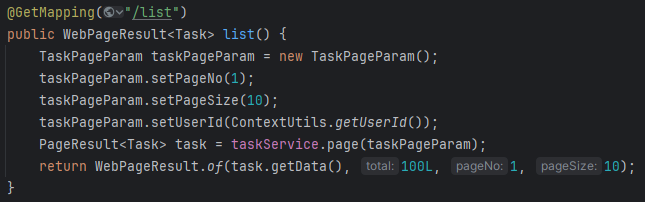


* **Objective:** To test if the **TestableOpenAIClient** masks an API key correctly.
* **Expected Result:** The test verifies that the **maskApiKey** method returns a masked version of the API key, where certain characters are replaced with asterisks to conceal the full key. The expected output is a string that correctly displays the masked API key.

## Mocking

Mocking in testing refers to the creation of fake objects that mimic the behavior of real ones, allowing developers to isolate the piece of code they’re testing. This technique simplifies the testing process by providing control over the responses of dependencies, ensuring tests run quickly and deterministically. Ultimately, mocking is invaluable for verifying that a unit of code correctly interacts with its external dependencies, without having to rely on those dependencies being present.

## Feature for Mocking: `WebPageResult.of()` Method



In the **TaskController** class, an additional feature suitable for mocking is the **WebPageResult.of()** method. This method creates a **WebPageResult** object for paginating tasks. Mocking this method offers distinct testing advantages by focusing on the controller's logic for preparing pagination parameters and verifying the correct interaction with pagination functionality.

## Advantages of Mocking WebPageResult.of()

* **Isolation of Controller Logic**: Mocking isolates the controller from the pagination logic, enabling tests to verify if the controller accurately sets up pagination based on the service response.
* **Behavior Verification**: It allows checking that **WebPageResult.of()** is called with correct arguments, ensuring the controller's pagination logic functions properly.

## How Mocking Enables Behavior Checking

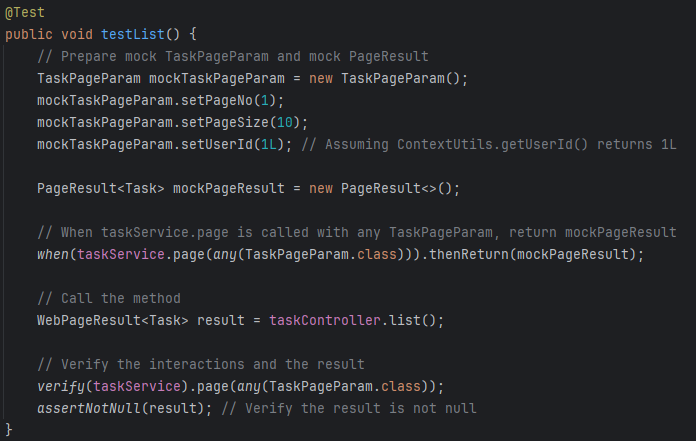
Mocking **WebPageResult.of()** facilitates behavior checking by:

* **Validating Parameters**: Ensures the controller properly uses the service layer's response for pagination.
* **Control Over Return Values**: Tests can manipulate the return value to simulate various pagination scenarios.
* **Decoupling from Implementation**: Makes tests more resilient to changes in the pagination logic or **WebPageResult** structure.

Mocking this method enhances tests by providing insights into how the controller processes pagination, beyond what direct testing offers.

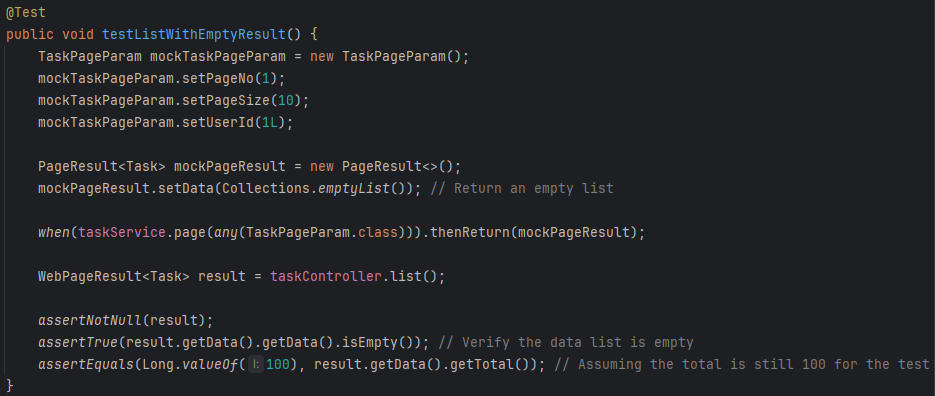
## Mockito Test case for: `WebPageResult.of()` Method

**Test Case 1: testList()**



* **Objective**: To verify that the **list()** method in **TaskController** correctly interacts with the **TaskService** to retrieve a page of tasks and returns a valid **WebPageResult<Task>** object. This test simulates the scenario where **TaskService** returns a non-empty list of tasks.
* **Expected Result**: The test checks if the **taskService.page()** method is called with the correct **TaskPageParam**. It also verifies that the **list()** method returns a non-null **WebPageResult<Task>** object. This ensures that the controller is correctly processing the service layer's response and packaging it into the expected web page result format.

**Test Case 2: testListWithEmptyResult()**



* **Objective**: To ensure that the **list()** method can handle scenarios where the **TaskService** returns an empty list of tasks. This test aims to simulate a situation where there are no tasks to display for the given page parameters.
* **Expected Result**: The test checks if the **list()** method still returns a valid **WebPageResult<Task>** object even when the list of tasks is empty. It specifically verifies that the data list within the result is empty and that the total count reflects the expected value (assumed to be 100 for the test). This confirms the controller's ability to handle empty data sets gracefully.

**Test Case 3: testListWhenServiceThrowsException()**



* **Objective**: To test the **list()** method's behavior when **TaskService** throws an exception. This scenario simulates an error condition within the service layer to assess the controller's error handling capabilities.
* **Expected Result**: The test expects a **RuntimeException** to be thrown when calling the **list()** method under this condition. By checking for the exception, it evaluates the controller's ability to propagate errors from the service layer, ensuring that error conditions are not silently ignored or improperly handled.

# Static Analyzers

## SpotBugs

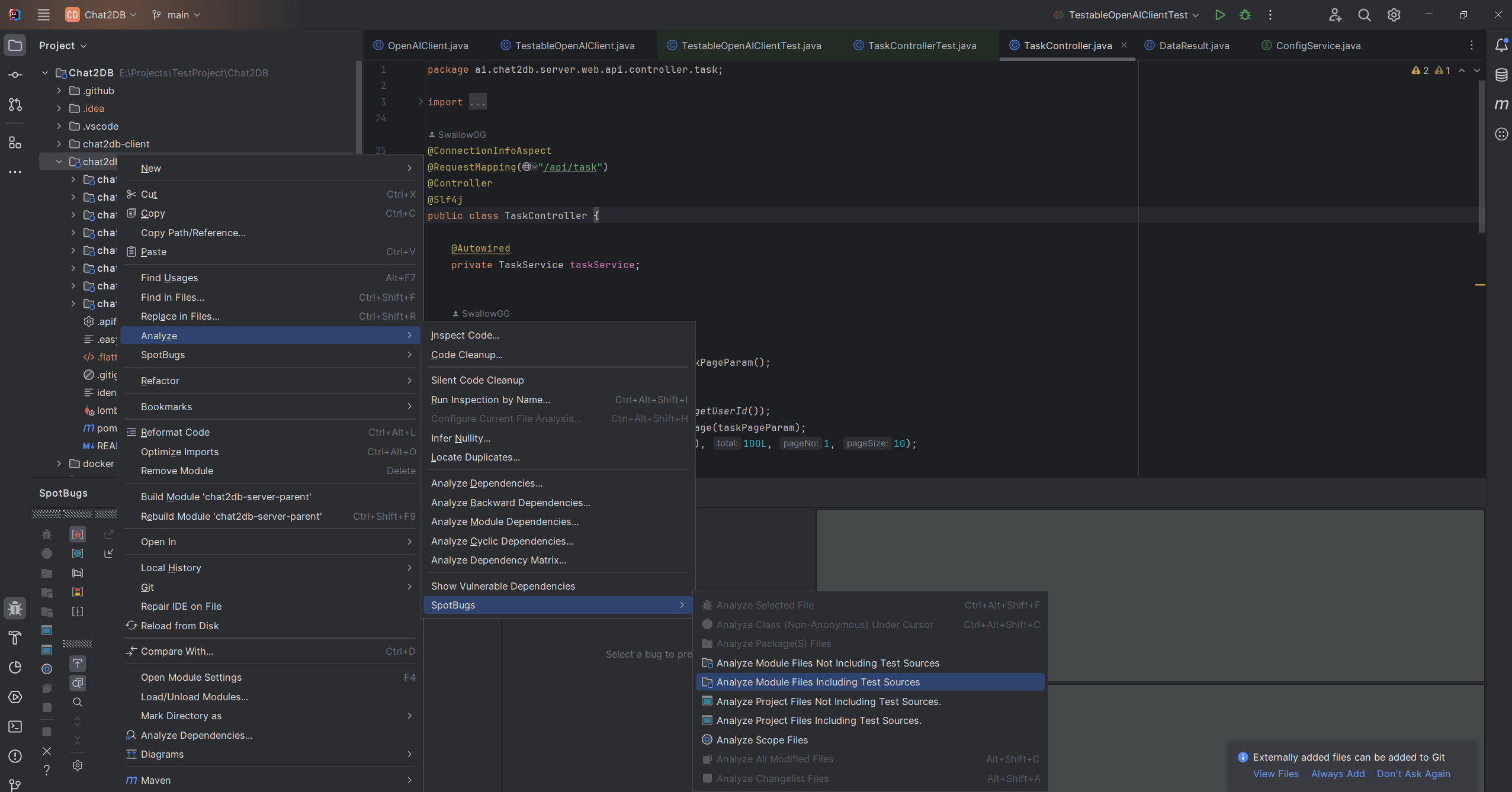
SpotBugs is a static code analysis tool that inspects your code for occurrences of bug patterns. Bug patterns are instances of common mistakes in programming that can lead to software vulnerabilities, performance issues, or outright crashes.

SpotBugs checks for:

* **Possible null pointer dereferences**
* **Incorrect behavior of equals() and hashCode()**
* **Bad practices that could lead to performance degradation**
* **Security vulnerabilities like SQL injection, hard-coded credentials**
* **Multithreading and synchronization issues.**

**In IDEA, in order to run SpotBugs:**

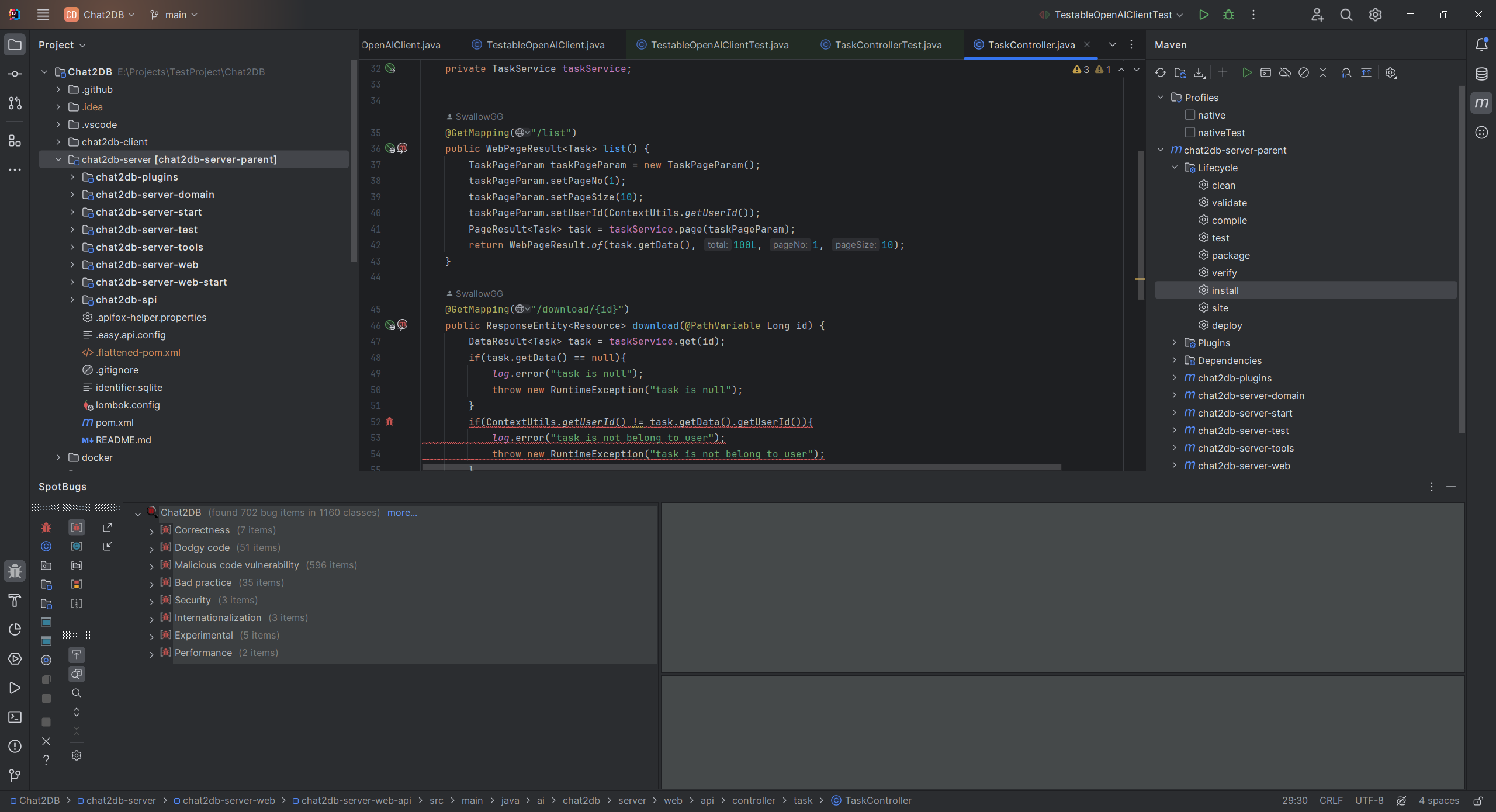
1. **Download the plugin from the Marketplace**
2. **Right-click on the project, module, package, or even a specific file in the Project View.**
3. **Select Analyze > SpotBugs > Choose one option from “Analyze Module Files Not Including Test Sources”, “Analyze Module Files Including Test Sources”, “Analyze Project Files Not Including Test Sources” and “Analyze Project Files Including Test Sources”.**



**Select an Option to Analyze**

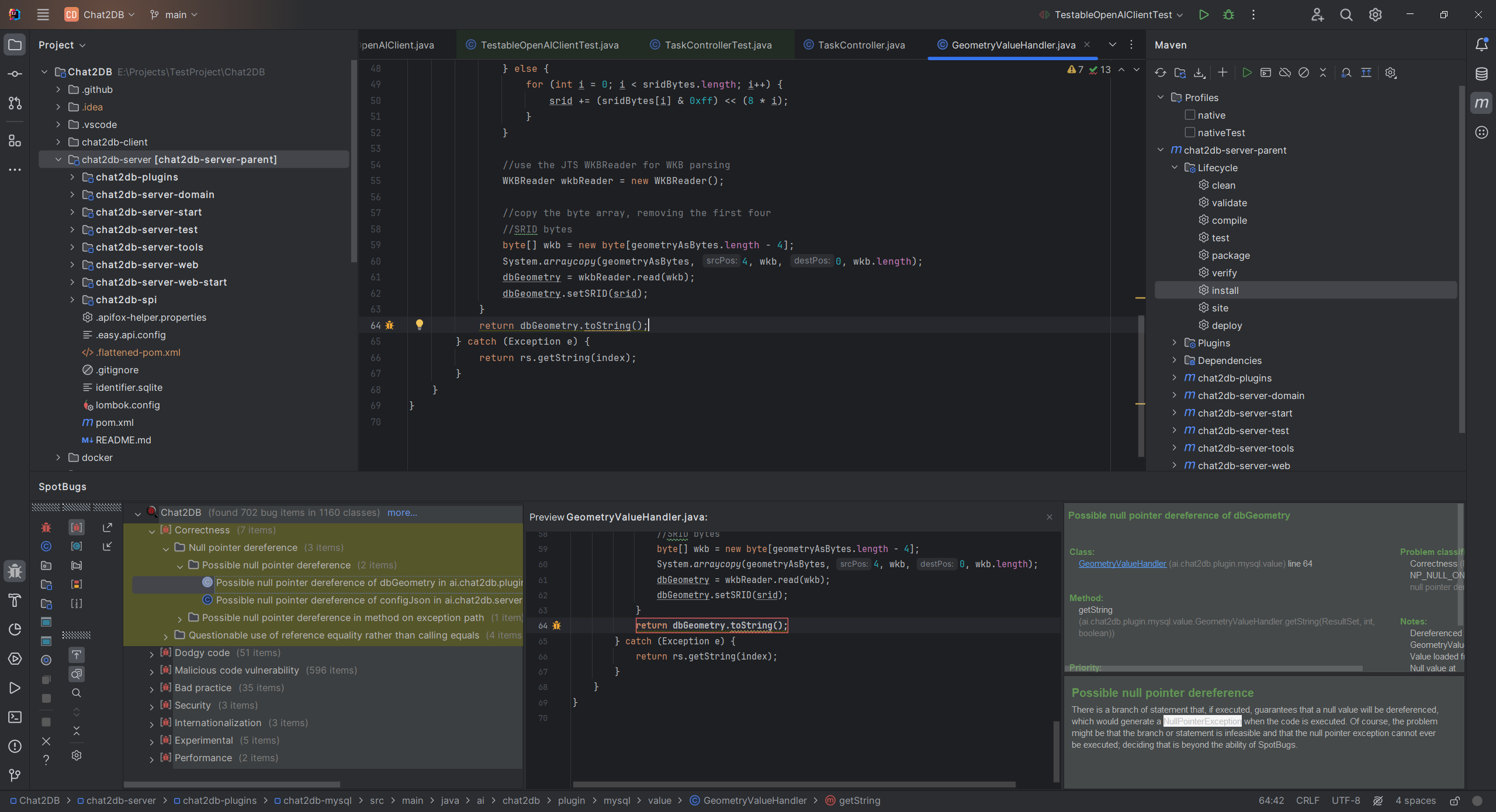
**From the results we can see there are 8 types of bugs, totally 702 bugs in 1160 classes:**

1. **Correctness: Issues that are likely to be bugs, such as incorrect calculations or logic, dead code, and other runtime errors.**
2. **Dodgy code: Code that is confusing, possibly wrong, or prone to errors. These might not be bugs, but they warrant a closer look to ensure they behave as expected. Examples include switch statements without a default case, unnecessarily complex or confusing expressions, and cases where the developer might have made an error in logic.**
3. **Malicious code vulnerability: These warnings are related to potential security flaws that could be exploited. This includes, for instance, code that might allow for SQL injection, code that exposes sensitive information, or code that could be abused for privilege escalation.**
4. **Bad practice: This category covers general coding practices that are not advisable because they make the code hard to maintain, prone to errors, or may degrade performance. Examples include misuse of API methods, inappropriate exception handling, and misuse of finalizers.**
5. **Security: These are potential vulnerabilities that could be exploited to compromise the security of the system. This might include things like hard-coded passwords, insecure random number generation, or predictable cryptographic keys.**
6. **Internationalization: Bugs in this category relate to the use of strings and character encoding, which can cause problems when software is used in different locales (geographic and cultural regions). For example, issues might arise from not handling Unicode characters properly, or incorrect string comparisons that do not consider locale settings.**
7. **Experimental: These are new or experimental bug patterns in SpotBugs. They might not be as reliable as the other categories, and could either indicate a real issue or be a false positive.**
8. **Performance: These issues can slow down the program's execution. They might involve inefficient use of collections, unnecessary object creation in loops, or other patterns that could be optimized for better performance.**



**Analysis Results**

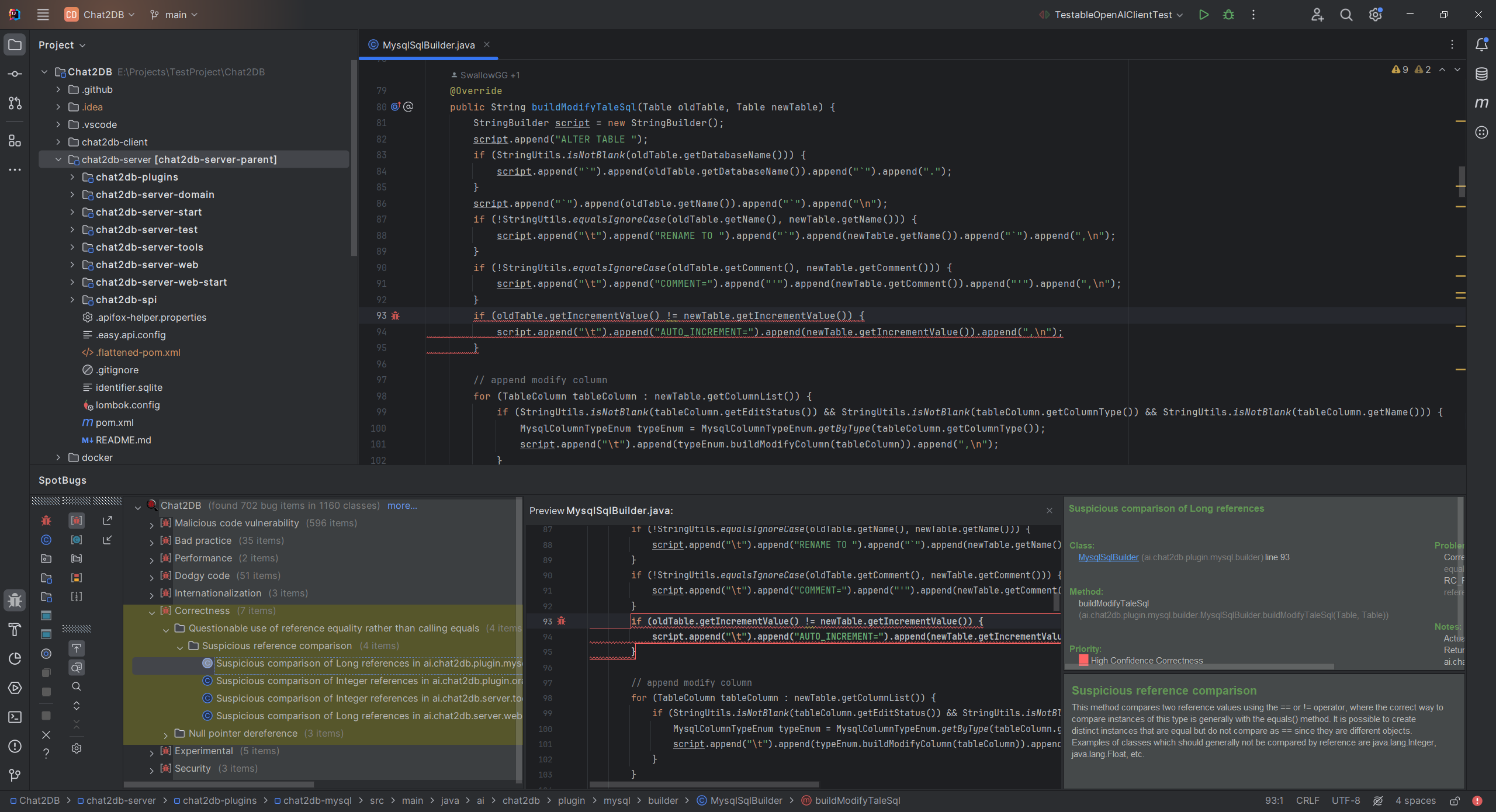
**In the code, bugs detected are also highlighted and explanations are provided.**



**Correctness Bug**

### Correctness Bug Example

**Suspicious Reference Comparison Bug**



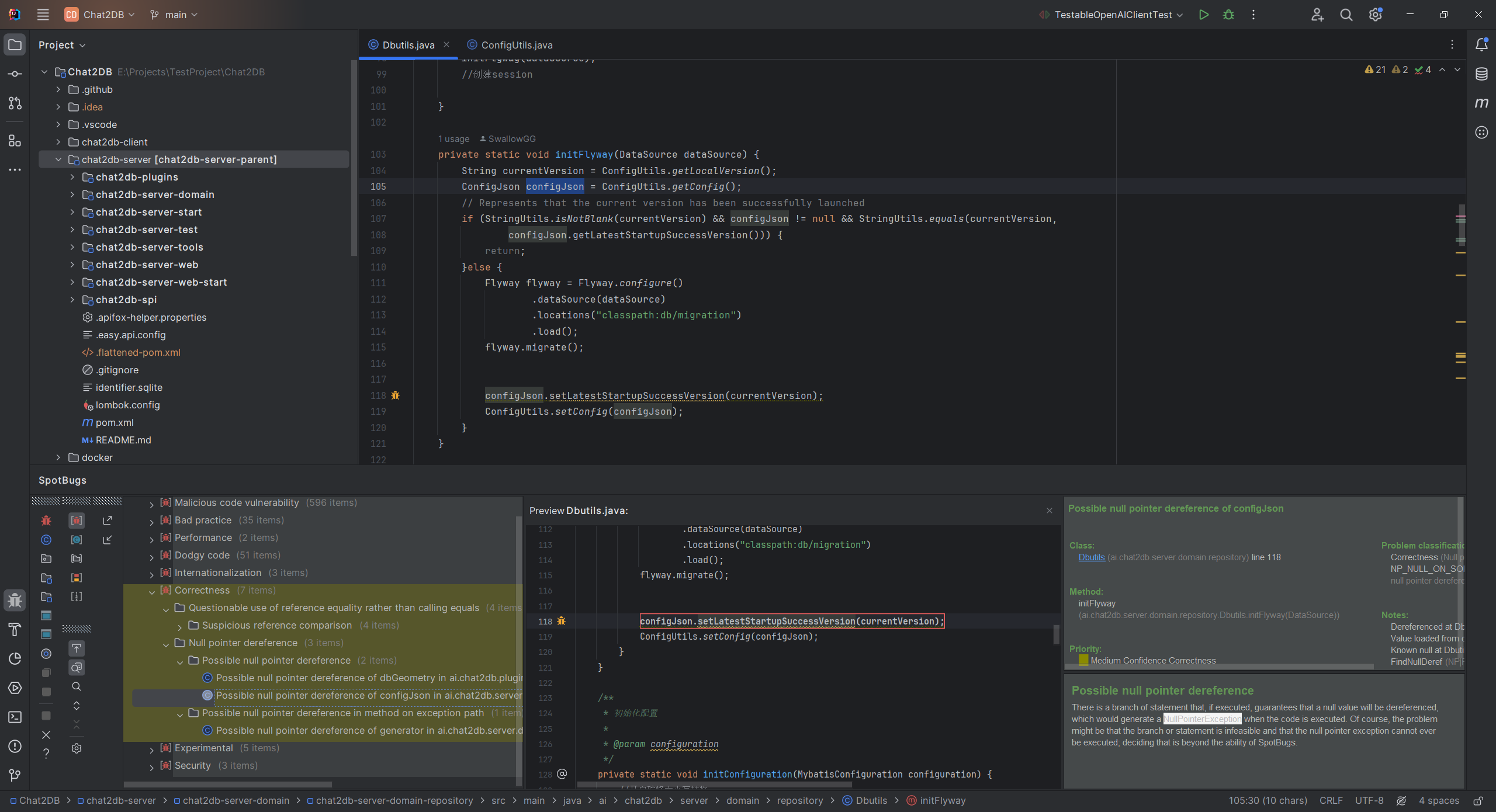
Suspicious Reference Comparison

According to the description, when instances of classes like Integer or Long are compared using “==” or “!=”, the distinct objects, instead of their values, are compared. Thus, two distinct objects that hold the same value and logically should be considered equal, are actually considered as different objects.

When we want to compare the values within two objects to see if they are logically equivalent (even if they are two separate instances), the **equals()** method should be used, which is designed to compare the content of objects.

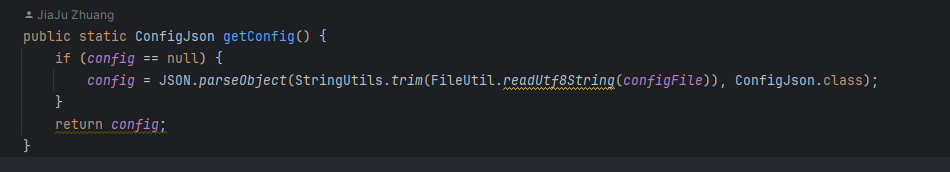
**Possible Null Pointer Dereference Bug**

In this example, the **oldTable** and **newTable** Objects are instances of **Table** class, and their **getIncrementValue()** method will return a Long Object. Considering the logic, the **equals()** method which compares the values should be used instead.



Possible Null Pointer Dereference

According to the description, the **configJson** could be a null value, which would generate a NullPointerException when the code is executed. However, checking the **getConfig()** in the **ConfigUtils** class, we could see the method has already makes sure the config is not null by checking it before returning.



The getConfig Method

Thus, this NullPointerException will never be generated and it is not an actual bug.

## PMD

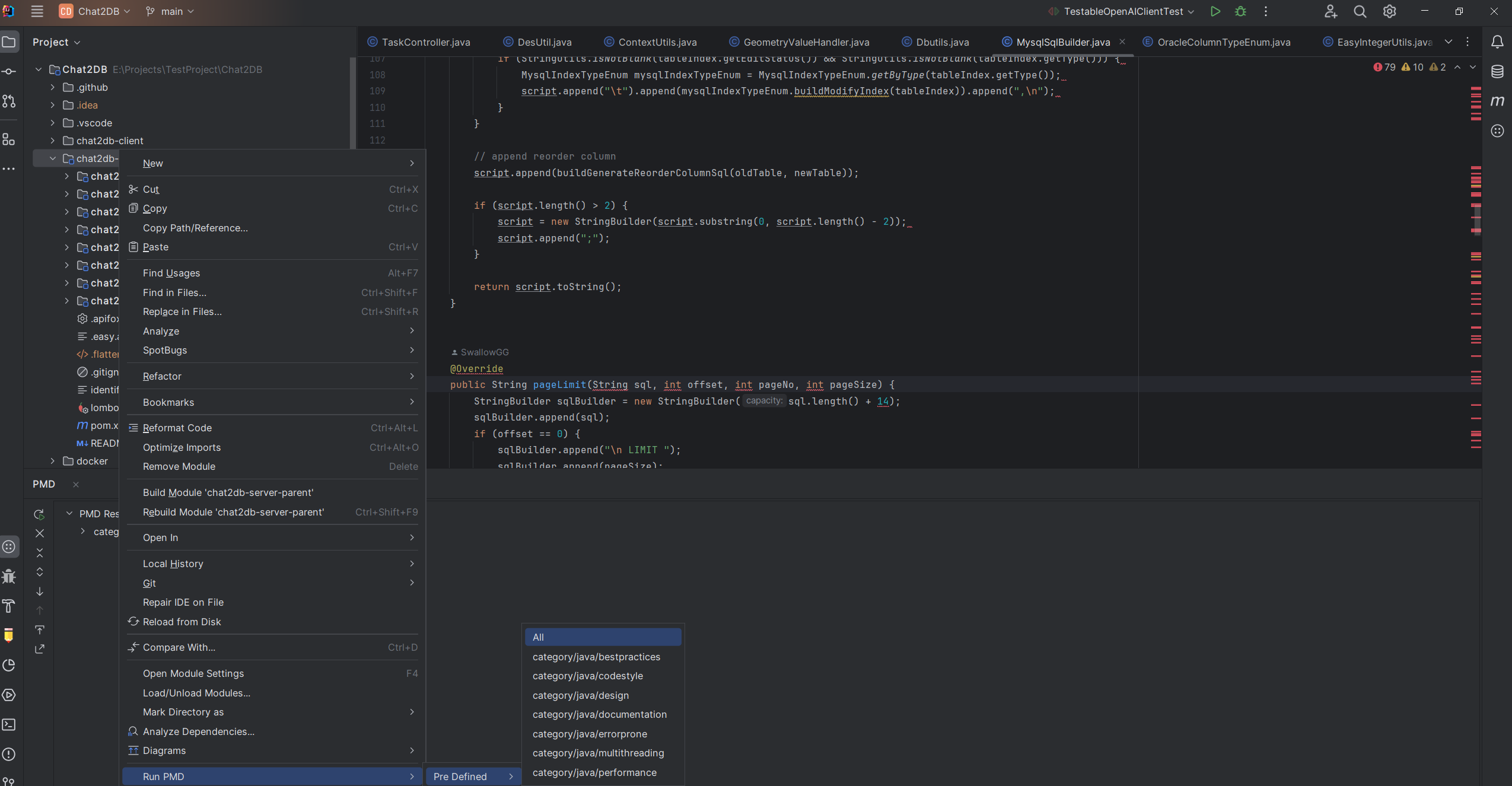
PMD is also a static code analysis tool that can scan Java source code and identify potential code problems.

PMD checks for:

* **Possible bugs: Empty try/catch/finally/switch blocks or unnecessary object creation, etc.**
* **Dead code: Unused local variables, parameters and private methods.**
* **Empty blocks: Blocks of code with no statements.**
* **Suboptimal code: Wasteful String/StringBuffer usage.**
* **Overcomplicated expressions: Unnecessary if statements, for loops that could be while loops.**
* **Duplicate code: Copied/pasted code means copied/pasted bugs, and this feature helps in identifying them.**

**In IDEA, in order to run PMD:**

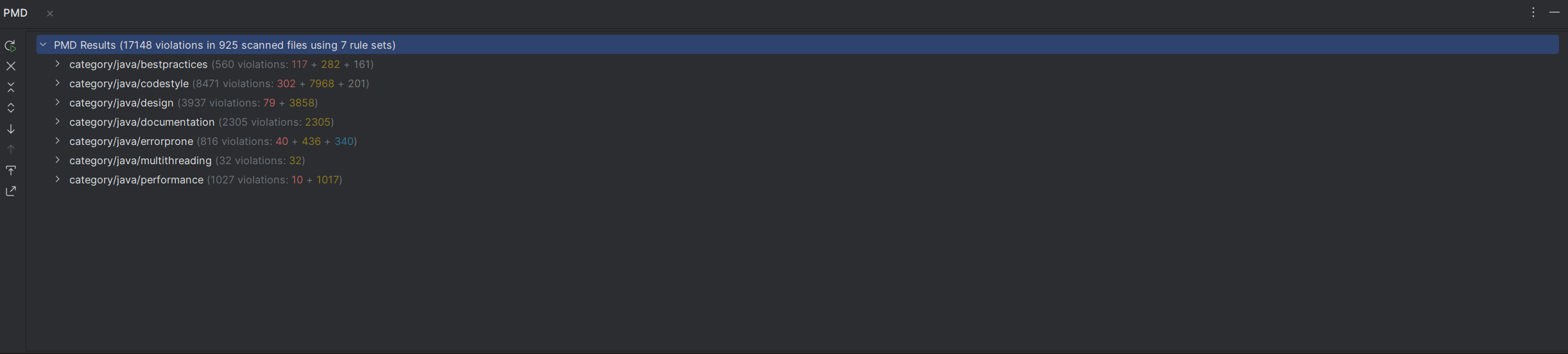
1. **Download the plugin from the Marketplace**
2. **Right-click on the project files**
3. **Select Analyze > Run PMD > Pre Defined > Choose one option from “All”, “**Best Practices**”, “**Code Style**”, “**Design**”, “**Documentation**”, “**Error Prone**”, “**Multithreading**” and “**Performance**”.**



**Select an Option to Analyze**

**Each of the options focuses on a particular kind of issue:**

1. Best Practices: Patterns that are generally considered good practice, often focused on maintainability and readability.
2. Code Style: Conventions on code formatting and structure which help to keep the code consistent and easy to read.
3. Design: Principles and patterns concerning software design, often aimed at avoiding overly complex or cumbersome designs and ensuring a clean architecture.
4. Documentation: Enforcement of proper commenting and documentation practices, ensuring that code is well-documented for future maintainers.
5. Error Prone: Code constructs that are often misused or likely to lead to errors, such as switch statements without a default.
6. Multithreading: Issues specific to concurrent programming, ensuring threads are managed safely without race conditions or deadlocks.
7. Performance: Identifying code that is inefficient and could slow down program execution, prompting the developer to consider more efficient alternatives.



**Analysis Results**

The PMD analysis detects 17148 violations in 925 scanned files. Of all violations, 548 are marked as errors.

### Bug Example

## Comparison