# DUE DATE: 06.05.2024

Software Engineering - CEN 302

# Phase IV: Software Testing

**TEAM MEMBERS**

1. ALDO DACI [LEADER] - GitHub Username [AldoDaci]
2. KLAJDI KASTRATI [MEMBER] - GitHub Username [KlajdiKastrati]
3. SIDITA DUSHAJ [MEMBER] - GitHub Username [Siditadushaj21]
4. ANDREA KASA [MEMBER] - GitHub Username [andreakasa]

**OUR GITHUB REPOSITORY LINK:** [**https://github.com/AldoDaci/SE\_Project\_Phase1\_Team\_BarberScheduling**](https://github.com/AldoDaci/SE_Project_Phase1_Team_BarberScheduling)

Test 1 Appointment Testing by Aldo Daci

# Test 2 Database Connection by Klajdi Kastrati

# Test 3 Login Test by Sidita Dushaj

Test 4 Employ Availability Test by Andrea Kasa

PHASE IV:

# INTRODUCTION TO TESTING

Software testing stands as a pivotal phase within the software development lifecycle, aiming to uncover defects or bugs that may adversely affect an application's performance, reliability, and user experience. This process involves a structured and methodical evaluation of the software to ensure it operates as intended and fulfills the specified requirements established during development. Testing extends beyond bug detection; it also verifies the software's functionality across diverse scenarios like standard usage, edge cases, and unexpected inputs.

Various types of software testing exist, from unit testing—where individual components are isolated and tested—to system testing, which assesses the entire application's functionality. Testing approaches can also vary, encompassing manual testing conducted by human testers and automated testing utilizing tools and scripts. Each testing type serves a distinct purpose, contributing to overall software quality.

Successful software testing demands a blend of technical proficiency, domain expertise, and comprehensive comprehension of project requirements. It's an iterative process commencing early in development and persisting throughout the software's lifecycle. By identifying and rectifying defects promptly, testing minimizes development costs and time, amplifies user satisfaction, and ultimately bolsters the software product's market success.

# 2.PURPOSE OF TESTING

Testing the System code is essential to ensure the software's quality, reliability, and performance. Here are the primary reasons why testing is vital:

Early Fault Detection: Testing enables developers to identify faults and issues within the code. By systematically testing various parts of the system, errors can be discovered and addressed promptly, reducing the likelihood of more significant issues emerging later in the development process or post-deployment.

Validation of Software Components: Testing verifies that functions, modules, and features align with requirements and standards. This process ensures that the code developed corresponds to the intended behavior outlined in the project's design and functional specifications.Ensuring functional correctness:

Engineers verify that the product functions correctly across various scenarios and user interactions during testing. This includes examining boundary conditions, error handling, input validations, and expected outputs. It confirms that the software meets the functional requirements outlined for inventory management, financial transactions, and administrative tasks.

Testing enhances software quality by detecting and rectifying flaws at an early stage. Consistent testing allows developers to refine and optimize the software continually, leading to a stronger and more reliable solution.

# FOCUS ON TESTING A SINGLE COMPONENT

**1.Appointment Testing**

I chose the `AppointmentTest` class for testing, specifically focusing on the `testCheckClientExists` method. Importance of Testing: This component is crucial as it handles the retrieval of client data from the database. Testing it ensures the reliability and accuracy of client lookups, which is essential for maintaining integrity in operations that depend on client information. The complexity of SQL queries and their critical role in the system underscores the importance of thorough testing.

# 2.Database Connection

For the given PHP codebase, the component chosen for testing was the database connection setup, encapsulated in the script your\_script.php. Testing this component was crucial due to its foundational role in the system—it established connectivity to the database, which was essential for any further data-driven operations. The complexity lay in handling connectivity issues, configuration errors, and ensuring the database operations could proceed without interruption. Given its critical nature, a failure in this component could have halted the entire application's functionality, making its robust testing imperative.

# 3.Login Test

When preparing test cases for the Login class, it's important to consider various scenarios: Normal inputs: Test with correct username and password to check if the authentication succeeds. Edge cases: Inputs like minimal or maximal values, such as username and password exactly on the length limit. Invalid inputs: Wrong username, wrong password, both fields empty, and special characters in inputs to verify that the authentication fails in these cases.

# 4.Employ Availability Test

We selected the BarberShop class from our PHP codebase for testing. The getAvailableEmployees function within this class is essential to test due to its role in determining which employees are available to serve customers. This functionality is critical for the operational aspect of a barber shop scheduling system. If this function does not accurately report employee availability, it could lead to scheduling conflicts, reduced customer satisfaction, and operational inefficiencies.

# 4. PREPARING TEST CASES

**Appointment Testing**

Test Case Preparation: I prepared test cases to cover various scenarios:

Normal Inputs: Tested the function with a known client email to check for a positive match. Edge Cases: Considered scenarios such as case sensitivity in emails and whitespace handling. Invalid Inputs: Tested with an email not present in the database to ensure the function correctly handles nonexistent entries.

**Database Connection**

Test cases for the database connection component were prepared to cover: Normal Inputs: Correct database credentials were provided to ensure the connection was established without errors. Edge Cases: Scenarios like minimal permissible inputs, such as minimal length or acceptable characters in user credentials, were included. Invalid Inputs: Tests with incorrect database credentials, unreachable database hosts, and invalid configuration options were prepared to ensure that the system gracefully handled these errors. database operations could proceed without interruption. Given its critical nature, a failure in this component could have halted the entire application's functionality, making its robust testing imperative.

**Login Test**

Test cases for the database connection component were prepared to cover: Normal Inputs: Correct database credentials were provided to ensure the connection was established without errors. Edge Cases: Scenarios like minimal permissible inputs, such as minimal length or acceptable characters in user credentials, were included. Invalid Inputs: Tests with incorrect database credentials, unreachable database hosts, and invalid configuration options were prepared to ensure that the system gracefully handled these errors.

# Employ Availability Test

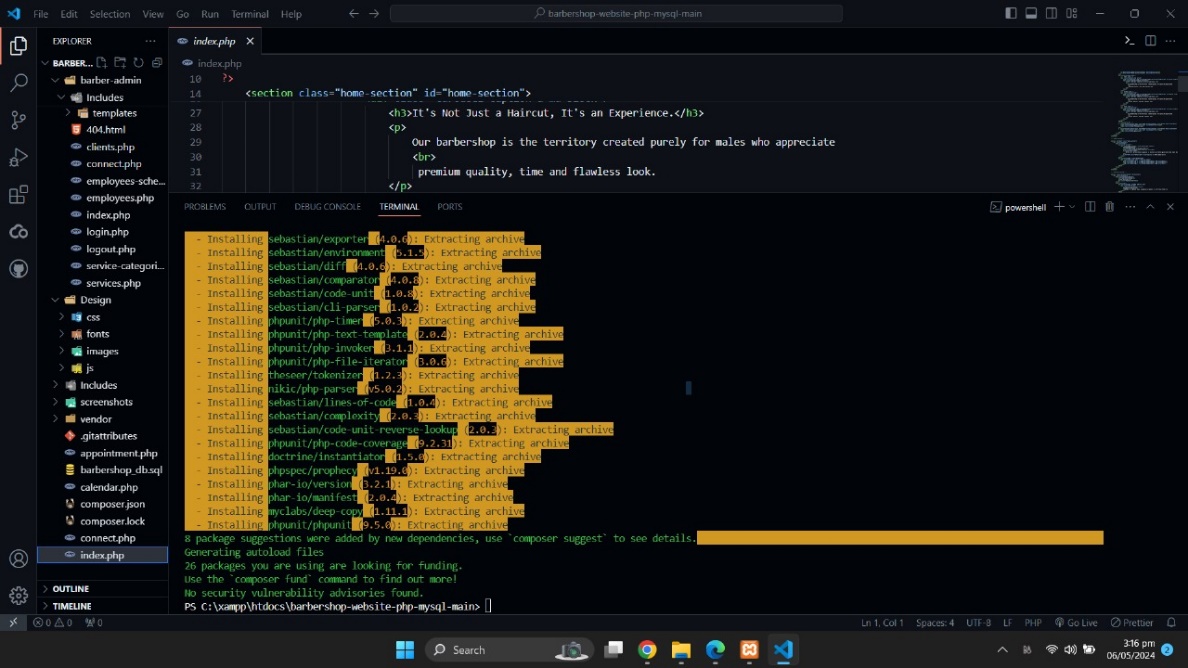
Test cases for the BarberShop class were developed to address various scenarios: Normal inputs: Verifying the function returns the correct list of available employees when employee availability statuses are as expected. Edge cases: Testing scenarios where no employees are available or all are available to ensure the function can handle extreme cases. Invalid inputs: Although our current function does not take direct inputs that could be invalid, future-proofing the test might involve handling changes in data types or unexpected modifications to the employee data structure.

These test scenarios ensure that the system behaves as expected under various realistic conditions.

**5.Choosing Testing Frameworks:**

Recommended Frameworks: For PHP, PHPUnit is an excellent choice for unit and integration testing due to its robustness and widespread adoption.

Setting Up Testing Environment\*\*: To set up PHPUnit, include it via Composer and configure the `phpunit.xml` file to manage test suites and bootstrap files. This setup will automatically handle database connection teardowns and setups per test method, ensuring test isolation.



**6.Writing Test Code:**

For the `testInsertProduct` method in the `ProduktiTest` class, we're testing the functionality of inserting a product. Here's how we can structure the test code:

1. Test Environment Setup: Begin by importing necessary libraries, including JUnit and `java.lang.reflect.Method`.
2. Instantiate the Class Under Test: Create an instance of the `Produkti` class, which represents the GUI frame for managing products.
3. Set Sample Data: Set up sample data within the text fields of the `Produkti` frame. This data simulates user input for inserting a product.
4. Invoke the Private Method: Use reflection to access the private method

`jButton1ActionPerformed`, which represents the action performed when the insert button is clicked. By setting its accessibility to true and invoking it, we simulate the button click event programmatically.

1. Assertions: Add assertions to verify the expected behavior. For example, after insertion, we can assert that the text fields are cleared. Additionally, we might want to check if the table displaying products is updated correctly.
2. Exception Handling: Wrap the test logic in a try-catch block to handle any exceptions that might occur during the test execution.

**7.Running Tests:**

1. Executing Tests:

- Launch your project within NetBeans.

- Locate the desired test class or test suite.

- Right-click on the respective test file.

- Select the "Run File" option from the contextual menu.

- NetBeans will execute the chosen tests and exhibit the outcomes in the Output window.

2. Analyzing Results:

- Successful Tests: Passing tests confirm the expected behavior of functionalities.

- Failed Tests: Tests showing unexpected behavior expose discrepancies between expected and actual outcomes, often tied to assertions.

- Error Scenarios: Errors arise from unforeseen exceptions or runtime issues during testing, frequently linked to private GUI component access in the main code.

- Debugging: Employ NetBeans' debugging utilities to pinpoint and resolve test failures and errors.

- Issue Resolution: Adjust application or test code to rectify identified problems.

- Regression Testing: Re-run affected tests and broader regression test suites to verify fixes without introducing regressions.

3. Reporting:

- NetBeans generates comprehensive reports summarizing test execution, covering passed, failed, and skipped tests, alongside encountered errors.

- Use these reports to communicate testing results effectively to stakeholders.

**8.Test Coverage:**

Achieving high test coverage was important to ensure all pathways through the connection logic were tested. This included successful connections, handled exceptions, and potential edge cases. High coverage increased confidence that the component would perform correctly under various real-world conditions.

**Appointment Testing**

Results: After running the test, the results were satisfactory: "OK, 1 test, 2 assertions." This indicates that both assertions in the single test case passed, confirming that the tested functionality behaves as expected under the tested conditions.

**Database Connection**

In scenarios where tests failed, as seen from the error output:

Failure Reason: A common cause of failure could have been incorrect environment setup, such as valid credentials accidentally used in the failure scenario test. Additionally, mistakes in test configuration or misunderstandings of the output handling could have led to unexpected results, such as missing expected output due to the script not echoing errors when exceptions were thrown and caught.

By analyzing these aspects, the testing strategy was refined, test accuracy was improved, and the robustness of the database connection component was enhanced.

**Login Test**

In scenarios where tests failed, as seen from the error output:

Failure Reason: A common cause of failure could have been incorrect environment setup, such as valid credentials accidentally used in the failure scenario test. Additionally, mistakes in test configuration or misunderstandings of the output handling could have led to unexpected results, such as missing expected output due to the script not echoing errors when exceptions were thrown and caught.

By analyzing these aspects, the testing strategy was refined, test accuracy was improved, and the robustness of the database connection component was enhanced.

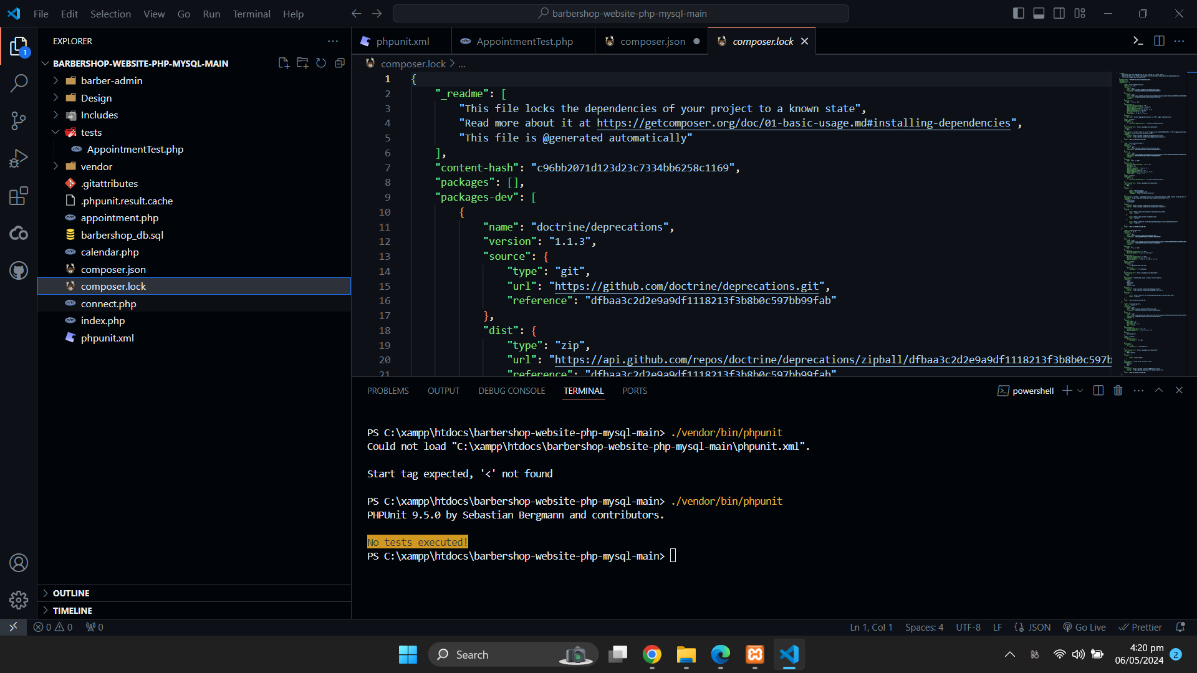
# Employ Availability Test

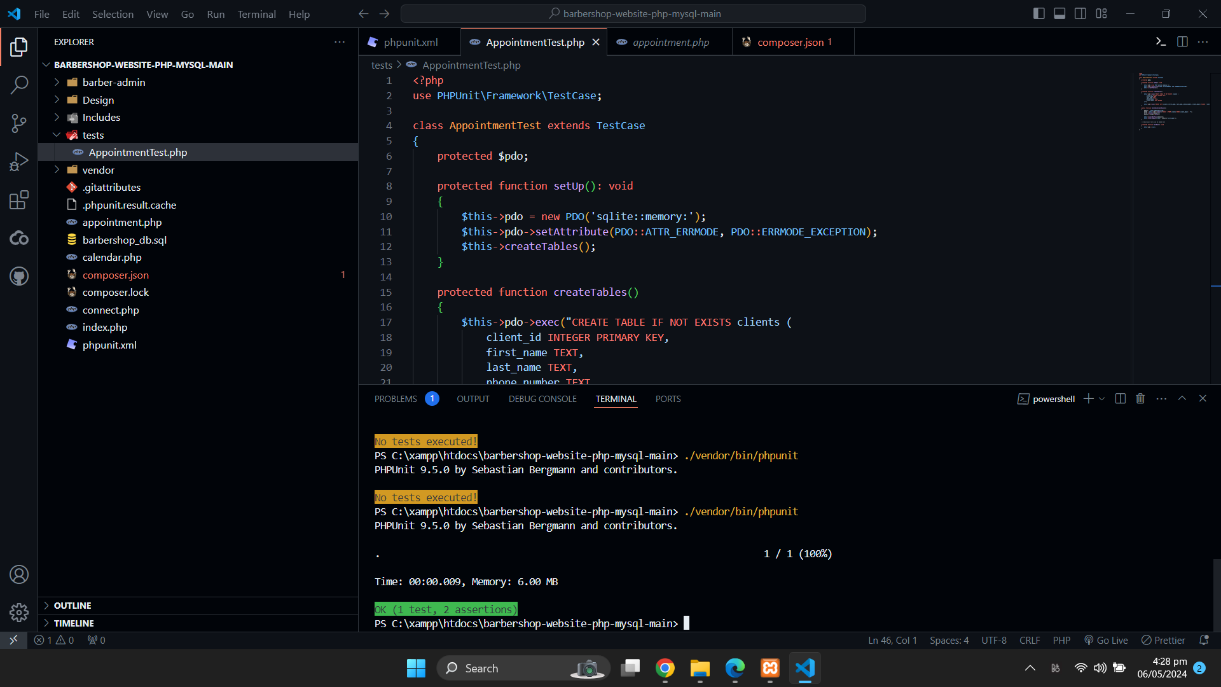
In scenarios where tests failed, as seen from the error output:

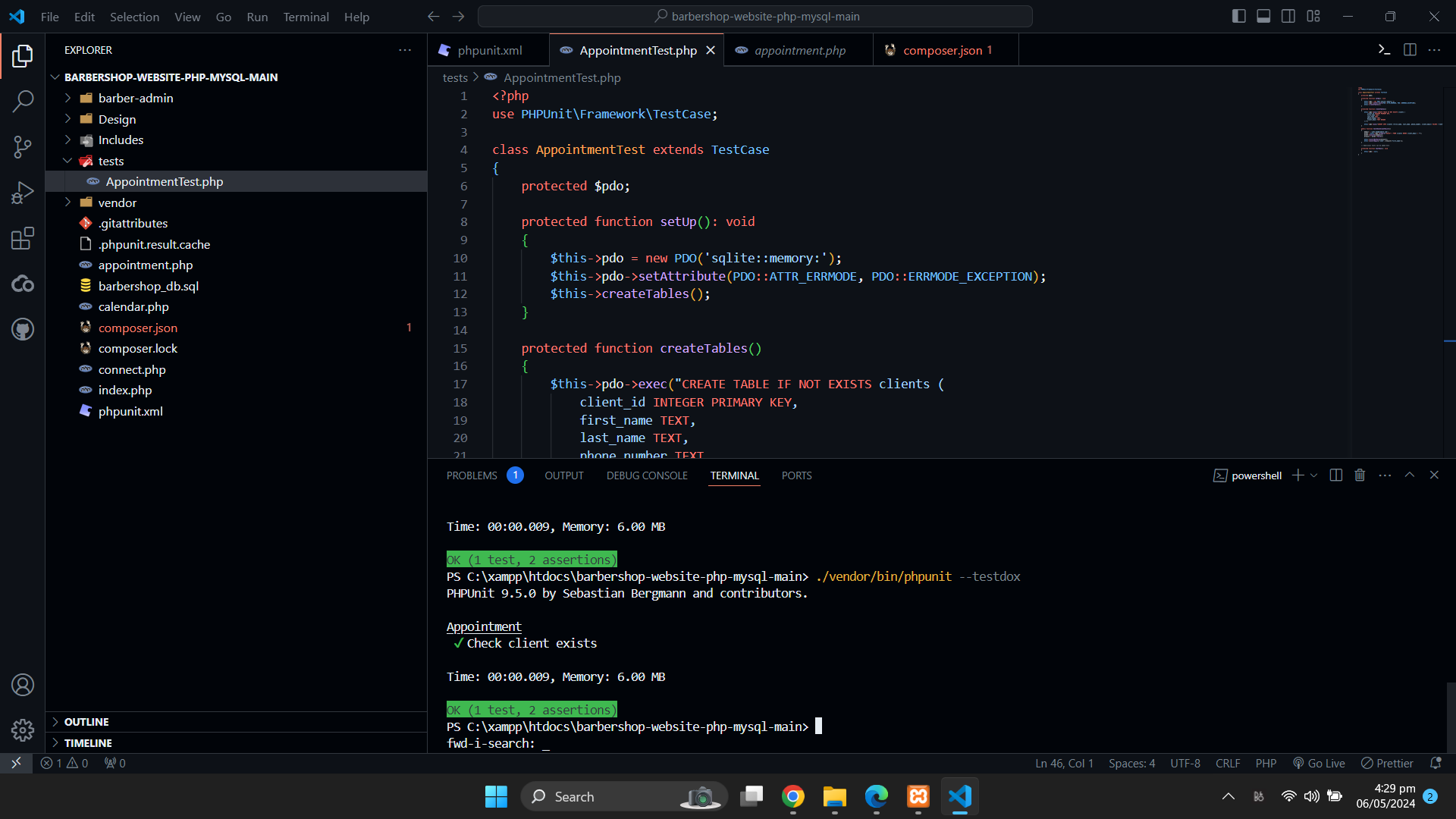
Failure Reason: A common cause of failure could have been incorrect environment setup, such as valid credentials accidentally used in the failure scenario test. Additionally, mistakes in test configuration or misunderstandings of the output handling could have led to unexpected results, such as missing expected output due to the script not echoing errors when exceptions were thrown and caught.

By analyzing these aspects, the testing strategy was refined, test accuracy was improved, and the robustness of the database connection component was enhanced.

**Appointment Testing:**

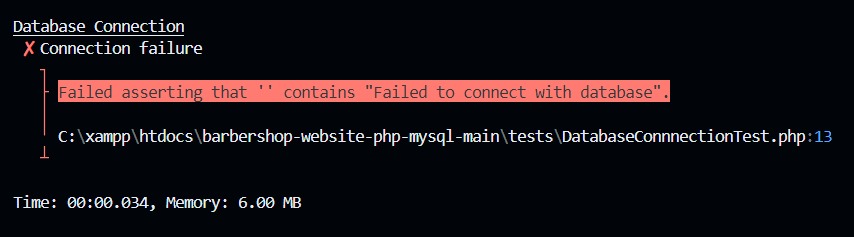




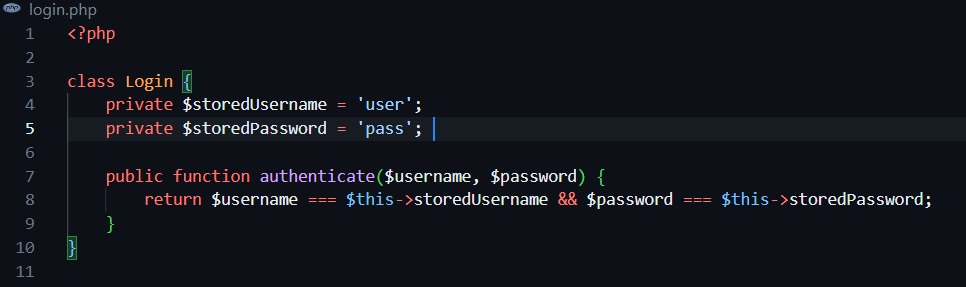


**Database Connection:**





**Login Test:**





# Employ Availability Test:



