Phase 4 Testing

1. **Introduction to Testing**

- Software testing is the process of evaluating software to identify defects or bugs. It involves running a program or system with the intent of finding errors, verifying that it meets specified requirements, and ensuring its reliability, correctness, and quality.

- Testing is crucial in software development as it helps to:

1.Ensure Reliability: Testing verifies that the software performs as expected under various conditions, reducing the likelihood of unexpected failures in real-world usage.
2.Ensure Correctness: Testing confirms that the software meets the specified requirements and behaves correctly, preventing potential issues that could arise due to incorrect functionality.

2. **Purpose of Testing**

- The purpose of testing is to identify defects early in the development process and verify that software components perform as intended. By doing so, testing ensures that potential issues are addressed promptly, leading to higher-quality software and reducing the risk of defects in the final product.

3. **Focus on Testing a Single Component**

Component: Attendance Tracking Screen

Importance of Testing:

- Data Accuracy: The Attendance Tracking screen is critical for recording employees' attendance data, including clock-in/out times, breaks, and leaves. Testing ensures that the screen accurately captures and updates attendance records, minimizing errors and discrepancies in payroll processing and scheduling.

- User Interface: The usability and intuitiveness of the Attendance Tracking screen impact user adoption and efficiency in recording attendance data. Testing assesses the screen's user interface design, navigation, and accessibility, ensuring a positive user experience for administrators and employees interacting with the system.

- Real-time Updates: Timely and accurate attendance data is essential for workforce management, scheduling, and decision-making. Testing verifies that the Attendance Tracking screen updates in real-time, providing instant visibility into employees' attendance status and facilitating proactive management of staffing needs.

- Error Handling: The Attendance Tracking screen should include robust error handling mechanisms to handle exceptions, such as invalid inputs or network connectivity issues. Testing evaluates the screen's error detection, validation, and recovery features, minimizing disruptions and ensuring data integrity in case of unforeseen circumstances.

- Security: Attendance data is sensitive and confidential, requiring adequate protection against unauthorized access or tampering. Testing validates the screen's security measures, including user authentication, encryption, and audit trails, safeguarding attendance records from unauthorized modifications or breaches.

4. **Preparing Test Cases**

Normal Inputs:
Test Case 1: Record attendance for an employee with valid In/Out times, employee name, type (e.g., Regular, Overtime), and message date.
Test Case 2: Update attendance for an employee who arrives and leaves on time, ensuring correct entry of all data fields.
Test Case 3: View the attendance log for a specific date and verify the presence of accurate attendance records for all employees.

Edge Cases:
Test Case 1: Record attendance for an employee with night shift hours spanning two consecutive days, ensuring correct handling of In/Out times and message date.
Test Case 2: Test the system's behavior when an employee works extended hours beyond standard working hours, verifying correct entry of Type (e.g., Overtime) and accurate message date.
Test Case 3: Verify attendance recording for part-time employees with non-standard working hours, ensuring correct handling of In/Out times and message date.

Invalid Inputs:
Test Case 1: Attempt to record attendance for an employee with missing or invalid data fields (e.g., missing employee name or In/Out times).
Test Case 2: Enter attendance data for a non-existent employee and ensure proper error handling.
Test Case 3: Submit attendance records with conflicting data (e.g., overlapping In/Out times) and verify error messages.

Boundary Conditions:

Test Case 1: Record attendance for an employee whose shift starts exactly at the beginning of the day (12:00 AM), ensuring correct handling of In/Out times and message date.
Test Case 2: Test the system's handling of maximum allowed working hours per day, ensuring accurate entry of Type (e.g., Regular) and message date.

Test Case 3: Verify attendance recording when an employee works a split shift spanning multiple time slots, ensuring correct entry of In/Out times and message date.

Error Handling:
Test Case 1: Test the system's response to network connectivity issues while recording attendance.
Test Case 2: Attempt to record attendance during scheduled system maintenance and verify system messages.
Test Case 3: Verify error handling when attempting to access the Attendance Tracking screen without proper permissions.

Concurrency and Performance:
Test Case 1: Simulate multiple administrators recording attendance for different employees simultaneously.
Test Case 2: Test the system's performance when generating attendance reports for a large number of employees.
Test Case 3: Assess the system's responsiveness under peak load conditions during the start/end of the workday.

Integration and Compatibility:
Test Case 1: Verify compatibility of the Attendance Tracking screen across different web browsers (Chrome, Firefox, Safari).
Test Case 2: Test the responsiveness of the screen on various devices (desktop, tablet, mobile).
Test Case 3: Verify integration with external timekeeping systems or biometric devices for automated attendance tracking.

Regression Testing:
Test Case 1: Re-run previously executed test cases after implementing updates or bug fixes to the Attendance Tracking screen.
Test Case 2: Verify that existing attendance records remain intact and accurate after system upgrades.
Test Case 3: Test critical functionalities affected by recent changes to the attendance recording process.

5 & 6 Choosing Testing Framework & Writing Test Cases

For Laravel applications, PHPUnit is the default choice for writing unit and integration tests. Laravel also provides convenient helpers and methods for testing various aspects of our application, including HTTP requests, database interactions, and authentication.

5.1 Setting up our environment

We need to install phpunit by the command below

composer require --dev phpunit/phpunit

5.2 Create Test Classes:

Laravel follows the convention of placing test classes inside the tests directory. We can generate test classes using Artisan commands:

php artisan make:test ExampleTest

5.3 Writing Test Cases

Once we have our test class generated, we can start writing test methods inside it. We will use Laravel's testing helpers and assertions provided by PHPUnit to write the test cases.

Below are three test cases for Attendance feature

```php
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;
use App\Models\Employee;
use App\Models\Attendance;

class AttendanceTest extends TestCase
{
   use RefreshDatabase;

   /**
    * Test recording employee attendance.
    *
    * @return void
    */
   public function testRecordAttendance()
   {
       // Create a test employee
       $employee = Employee::factory()->create();

       // Send a POST request to record attendance
       $response = $this->post('/attendance', [
           'employee_id' => $employee->id,
           'clock_in' => '2024-05-07 09:00:00',
           'clock_out' => '2024-05-07 17:00:00',
```

```
        ]);

        // Assert that the request was successful
        $response->assertStatus(200);

        // Assert that the attendance record was created
        $this->assertDatabaseHas('attendances', [
            'employee_id' => $employee->id,
            'clock_in' => '2024-05-07 09:00:00',
            'clock_out' => '2024-05-07 17:00:00',
        ]);
    }

    /**
     * Test retrieving attendance records for a specific employee.
     *
     * @return void
     */
    public function testRetrieveAttendanceRecords()
    {
        // Create a test employee
        $employee = Employee::factory()->create();

        // Create test attendance records for the employee
        Attendance::factory()->count(3)->create(['employee_id' => $employee->id]);

        // Send a GET request to retrieve attendance records
        $response = $this->get('/attendance/employee/' . $employee->id);

        // Assert that the request was successful
        $response->assertStatus(200);

        // Assert that the response contains the expected attendance records
        $response->assertJsonCount(3);
    }

    /**
     * Test error handling when recording attendance with invalid data.
     *
     * @return void
     */
    public function testRecordAttendanceWithInvalidData()
```

```
    {
        // Send a POST request to record attendance with invalid data
        $response = $this->post('/attendance', [
            // Missing required 'employee_id' field
            'clock_in' => '2024-05-07 09:00:00',
            'clock_out' => '2024-05-07 17:00:00',
        ]);

        // Assert that the request failed due to validation errors
        $response->assertStatus(422)
                 ->assertJsonValidationErrors(['employee_id']);
    }
}
```

5.4 Running the Tests

To run the test we can use the below command:

`./vendor/bin/phpunit`

7. Running Tests

1. **Passing Tests**: If a test passes successfully, PHPUnit displays a green bar indicating the number of tests executed and their duration. Each dot represents a passing test.
2. **Failing Tests**: If a test fails, PHPUnit provides detailed information about the failed assertion(s), including the expected and actual outcomes. It displays a red bar and highlights the failed test(s).
3. **Error Scenarios**: If an error occurs during test execution (e.g., syntax error, runtime exception), PHPUnit reports the error and its stack trace. It distinguishes errors from failures, indicating issues unrelated to test assertions.
4. **Output:** PHPUnit displays detailed output for each test method, including the test description, executed assertions, and any additional output generated by the test method.

Understanding Assertion Results:

1. When writing test methods, use PHPUnit's assertion methods (e.g., assertTrue(), assertEquals()) to validate expected outcomes.
2. If an assertion succeeds, PHPUnit continues executing subsequent assertions within the test method.

3. If an assertion fails, PHPUnit stops executing the current test method and marks it as failed. It reports the failed assertion and continues with other test methods.
4. Review the assertion results to identify which tests passed, failed, or encountered errors. Use this information to debug and fix issues in your codebase.

8. Test Coverage

Achieving high test coverage is crucial for ensuring thorough testing of software and comes with several important benefits:

- Identifying Uncovered Code Paths: Test coverage analysis helps identify areas of the codebase that are not exercised by existing tests. Uncovered code paths are potential areas where defects or bugs may exist but remain undetected due to lack of test coverage.
- Detecting Regression Issues: Comprehensive test coverage reduces the risk of introducing regressions when making changes to the codebase. Improving Code Quality: Writing tests often leads to writing better, more modular, and more maintainable code. When developers write tests for their code, they tend to design their code in a way that is easier to test, which often translates to cleaner and more organized code. Additionally, writing tests encourages developers to consider edge cases, error handling, and boundary conditions, leading to higher-quality code.
- Enhancing Confidence in Changes: High test coverage provides confidence when making changes to the codebase. Developers can refactor or modify code with confidence, knowing that existing tests will catch any unintended side effects or regressions.
- Facilitating Collaboration: Test coverage serves as a form of documentation for the behavior of the software. When new developers join a project or when existing developers revisit code they haven't touched in a while, comprehensive tests help them understand how different components of the software interact and behave.