Project Proposal - Version 1.0

**Optimizing Java Selenium Framework with Automated Test Distribution and Full Resource Utilization.**

Prepared for: John Doe, VP of QA

Prepared by: Siva Teja Mundlamuri

Team Lead

WeThink Technology Solutions

11/20/2023.

**Executive Summary:**

The "Optimizing Java Selenium Framework with Automated Test Distribution and Full Resource Utilization" project is a strategic initiative aimed at enhancing the efficiency and reliability of our automated testing processes. As web applications grow in complexity, the need for rapid and dependable testing becomes paramount. The current Selenium Java framework, while functional, does not fully capitalize on system resources, resulting in prolonged test execution times and delayed deployments. Manual test case distribution also poses inefficiencies, leading to resource wastage and increased hardware costs.

Under the leadership of John Doe, VP of Quality Assurance, this project aims to address these challenges by optimizing the existing Selenium Java framework. The primary objectives include refactoring test cases to support dynamic thread allocation, reducing test execution times by at least 30%, maintaining or improving test case reliability, and ensuring scalability for future testing needs.

The optimized framework will incorporate several key features, including the ability to read test cases from a database, support parallel execution with user-defined thread configurations, interact with databases for data retrieval and storage, distribute test cases based on system load, and generate consolidated test execution reports.

The project's business value is substantial, with the potential to accelerate speed to market, optimize resource allocation, and reduce costs.

However, the project faces challenges related to legacy code adaptability, the need to adhere to a predefined timeline without disrupting ongoing operations and ensuring comprehensive and accessible reporting. A detailed feasibility analysis has been conducted to evaluate the project's technical, operational, economic, legal, and security feasibility.

This project represents a strategic investment in optimizing our testing processes, aligning them with industry best practices, and delivering tangible benefits in terms of efficiency, cost savings, and speed to market.

**Contents:**

1: Introduction

1.1: Background on company

1.2: Problem Description

2: System Proposal:

2.1: System Request

2.2: Workplan

2.3: Feasibility Analysis

2.4: Verb/Noun Textual Analysis to determine Use-Case Actors and Methods

2.5: Requirements Definition

2.5.1: Functional Requirements

2.5.2: Non-Functional Requirements

3: Functional Model

3.1: Use-Case descriptions

3.2: Use-Case diagram

3.3: Activity diagram

4: Structural Model:

4.1: Class diagrams

4.2: Object diagram

5: Behavioural Model

5.1: Sequence diagram

5.2: Words connecting the structural and behavioural models on how I generated them from my functional models

6: Design Model

6.1: Package diagram

6.2: Contracts

6.3: Method Specifications

7: Data Management Layer Design

7.1: Object persistence format

7.2: Database Schema

8: Human-Computer Interaction Layer Design

8.1: Wireframe design of Test Report

9: Physical Architecture Layer Design

9.1: Hardware and Software Specifications Overview

9.1.1: Standard Client Specifications:

9.1.2: Standard Database Server Specifications

9.2: Operational Requirements

9.3: Performance Requirements

10: Construction

10.1: Unit Tests

10.2: System Requirements Testing

10.3: System Performance Testing

11: Acknowledgement

12: Conclusion

13: References

**1: Introduction:**

**1.1: Background on company:**

WeThink Technology Solutions is a leading software development and quality assurance firm dedicated to delivering high-quality digital solutions to our clients. With a focus on innovation and excellence, we have built a strong reputation for providing cutting-edge technology services, including software development, automation testing, and quality assurance.

John Doe is a highly respected professional in the field of quality assurance and testing. With 22 years of experience in the industry, John has played a pivotal role in ensuring the quality and reliability of our software products. As the VP of Quality Assurance, John is dedicated to optimizing our testing processes and tools to meet the growing demands of our clients and maintain our commitment to excellence.

As the complexity of our web applications grows, it has become increasingly critical to have efficient and reliable automated testing in place. The current Selenium Java framework is not fully utilizing system resources, leading to longer test run times and delayed deployments. To address this need, WeThink Technology Solutions is proposing the implementation of an optimized Selenium framework that maximizes resource utilization and minimizes test execution time and ensure the seamless distribution of test cases based on system load in real time.

**1.2: Problem Description:**

As the complexity of our web applications continues to grow, the need for efficient and reliable automated testing has become increasingly critical. However, our current Selenium Java testing framework faces several challenges that hinder its effectiveness:

Underutilization of System Resources: The existing framework does not fully leverage the available system resources, resulting in longer test execution times. This inefficiency leads to delayed deployments and hampers our ability to respond swiftly to market demands.

Manual Test Case Distribution: Test case distribution is currently a manual and resource-intensive process. This method is error-prone and leads to suboptimal resource utilization and increased hardware costs.

Lack of Scalability: The current framework lacks scalability to accommodate the growing number of test cases associated with our expanding web application portfolio. This limitation constrains our ability to meet future testing requirements effectively.

Data Management Challenges: Efficient data management before and after test execution is essential. The current framework lacks the capabilities to seamlessly interact with databases for data retrieval and storage, hindering the effectiveness of our test cases.

Inefficient Reporting: Test execution reports are fragmented and do not provide a unified view of test results. This makes it challenging for stakeholders to assess overall application quality.

To address these challenges, the "Optimizing Java Selenium Framework with Automated Test Distribution and Full Resource Utilization" project aims to refactor our Selenium Java framework, introducing dynamic thread allocation, parallel execution, database interaction for data management, and load-based test case distribution. These enhancements will optimize resource utilization, reduce test execution times, improve reliability, and enhance scalability. Furthermore, the project will deliver a consolidated test execution report for better visibility into test outcomes.

By tackling these issues head-on, we aim to streamline our testing processes, accelerate speed to market, optimize resource allocation, and reduce costs, ultimately strengthening our competitive edge in the market.

**2: System Proposal:**

**2.1: System Request**- Optimizing Java Selenium Framework with Automated Test

Distribution and Full Resource Utilization.

**2.1.1:** **Project Sponsor:** John Doe, VP of Quality assurance

**2.1.2: Business Need:**

As the complexity of our web applications grows, it has become increasingly critical to have efficient and reliable automated testing in place. The current Selenium Java framework is not fully utilizing system resources, leading to longer test run times and delayed deployments and Automated Test Distribution to optimize the distribution of test cases to multiple systems for efficient and cost-effective test execution. Currently, manual distribution of test cases is inefficient, leading to resource wastage and increased hardware costs.

**2.1.3: Business Requirements:**

2.1.3.1: Optimize Existing Tests: The existing Selenium Java test cases must be refactored to support dynamic thread allocation.

2.1.3.2: Speed and Efficiency: The optimized framework should aim to reduce the test execution time by at least 30%.

2.1.3.3: Reliability: Ensure that the optimized test cases are as reliable as the current ones, if not more.

2.1.3.4: Scalability: The framework should be scalable to accommodate future test cases.

2.1.3.5: Test Case Management: The system should be able to read test cases from a Database, including test case details and dependencies.

2.1.3.6: Parallel Execution: The framework should generate threads for each test case, enabling parallel execution based on a user-defined number of concurrent threads.

2.1.3.7: Data Management: It should interact with a database to retrieve and store data required for test case execution. This includes data retrieval before test execution and data storage after execution.

2.1.3.8: Load-Based Distribution: The system should distribute test cases dynamically to node machines based on system load. The distribution should be optimized to ensure efficient resource utilization.

2.1.3.9: Common Consolidated Report: A consolidated test execution report should be generated on the hub machine, regardless of which node executed the test cases. This report should provide a unified view of test results.

**2.1.4: Business Value:**

* Speed to Market: Parallel execution will reduce test execution time, enabling faster feedback and development cycle times, enabling quicker releases.
* Resource Optimization: The framework will optimize the allocation of test cases, saving human resources from manual distribution efforts.
* Cost Reduction: By distributing test cases based on system load, unnecessary hardware expenses will be reduced, resulting in cost savings.

**2.1.4.1: Expenditure:**

2.1.4.1.1: Development Cost: Estimated 200 hours at $100/hour = $20,000

* This is the cost for the actual development work to optimize the Java Selenium framework. This cost includes activities like planning, coding, code reviews, testing, and deployment of the optimized framework.

2.1.4.1.2: Miscellaneous Cost: $3,000

* Documentation: Costs associated with documenting the process and changes made during the project.
* Training: Costs for training the team on how to use the new optimized framework and monitoring tools.
* Contingencies: Any unexpected costs that may arise during the project, such as debugging difficult issues or addressing unforeseen challenges.

Total Cost: $23,000.

**2.1.4.2: Savings and Earnings:**

2.1.4.2.1: Hardware/Cloud Savings: Estimated $5,000/year

* By optimizing the framework to fully utilize system resources, there's a reduced need to invest in additional hardware or cloud computing power for test execution. This results in an estimated annual saving of $5,000. This figure takes into account the reduction in the need for virtual machines, servers, or cloud compute instances specifically designated for running Selenium tests.

2.1.4.2.2: Time Savings: Estimated 300 hours/year, valued at $30,000

* Optimizing the framework not only makes tests run faster but also reduces the amount of human intervention required for debugging, monitoring, and maintaining the test setup. The time saved is estimated to be 300 hours per year. If we value this time at $100 per hour (the assumed rate for skilled labor involved in this project), this translates to $30,000 saved annually.

2.1.4.2.3: Increased Revenue from Faster Time to Market: Estimated $50,000/year.

* An optimized, faster testing cycle enables quicker release times for new features, updates, or bug fixes. Being faster to market can offer a competitive advantage, potentially leading to increased revenue. The estimated additional annual revenue, in this case, is $50,000. This figure is highly dependent on the nature of the business and market conditions, and it may require in-depth analysis to arrive at an accurate projection.

Total Estimated Annual Savings and Earnings: $85,000/year

**2.1.5: Special Issues or Constraints:**

2.1.5.1: Legacy Code: Older test cases may not be easily adaptable to the new framework.

* The existing Selenium Java framework may have test cases that were developed some time ago and are now considered "legacy" code. These test cases might have been developed under different coding standards, or they may be using outdated libraries and methodologies that are not easily compatible with the optimized framework. Retrofitting these older test cases to fit the new optimizations may require extra time and resources, potentially affecting the project's timeline and budget.

2.1.5.2: Timeline: The optimization needs to be done without affecting the current release cycle.

* The urgency to not disrupt the ongoing development and release cycles adds another layer of complexity to the project. Changes to the testing framework could potentially introduce bugs or delays that affect other teams and projects. To minimize the impact, the transition to the optimized framework may have to be phased or scheduled during low-activity periods.

2.1.5.3: Reporting: Ensure that the consolidated test execution report is comprehensive and accessible to relevant stakeholders.

**2.2: Workplan:**

2.2.1: Project Initiation and Feasibility Analysis (Duration: 1 week):

* Define project objectives, scope, and stakeholders.
* Establish project team and roles.
* Conduct a kick-off meeting.
* Review the existing Selenium Java framework and identify areas for optimization.
* Assess the compatibility of legacy test cases with the new framework.
* Evaluate the impact of framework changes on existing development and release cycles.
* Conduct a risk assessment to identify potential challenges.
* Review the available budget and resources.

2.2.2: Requirement Analysis and Documentation (Duration: 1 weeks):

* Collaborate with business analysts and stakeholders to gather detailed requirements.
* Document functional and non-functional requirements.
* Create use cases and user stories.

2.2.3: Development (Duration: 8 weeks):

* Design framework architecture.
* Implement the Selenium Java framework.
* Integrate the framework with the Selenium Grid for parallel execution.
* Develop database interaction modules for test case management.
* Create reporting mechanisms for consolidated test results.
* Continuously update documentation as features are developed.

2.2.4: Testing and Quality Assurance (Duration: 2 weeks):

* Develop test cases for each requirement.
* Conduct unit testing, integration testing, and system testing.
* Perform performance testing to validate the framework's efficiency how many testcases and VM’s it can handle without deviations in performance.
* Address and resolve any identified defects.
* Ensure that the framework meets scalability requirements.
* Validate the framework's compatibility with different browsers and operating systems.

2.2.5: User Training (Duration: 1 weeks):

* Develop training materials and documentation for users.
* Train the testing team and relevant stakeholders on using the new framework.

2.2.6: Deployment and Monitoring (Duration: 1 week):

* Monitor the transition to the new framework.
* Address any issues or challenges that arise during deployment.

2.2.7: Project Closure (Duration: 1 week):

* Review the project against initial objectives and requirements.
* Obtain stakeholder feedback and conduct a post-project review.
* Update project documentation with lessons learned.
* Prepare a final report and presentation for stakeholders.
* Officially close the project.

**2.3: Feasibility Analysis:**

2.3.1: Technical Feasibility:

* Assess the technical challenges of optimizing the Selenium Java framework.
* Evaluate the compatibility of legacy test cases and the feasibility of retrofitting them.
* Ensure that the framework can fully utilize system resources (number of threads we have assigned for a particular system) without causing instability.

2.3.2: Operational Feasibility:

* Evaluate the impact of framework changes on ongoing operations, including development and release cycles.
* Consider the feasibility of phasing or scheduling changes during low-activity periods to minimize disruption.
* Ensure that the framework's reporting align with operational needs.

2.3.3: Economic Feasibility:

* Compare the estimated development cost of $23,000 with the projected annual savings and earnings of $85,000.
* Assess the return on investment (ROI) and payback period.
* Consider the cost of potential risks and contingencies.

2.3.4: Legal and Security Feasibility:

* Ensure that the handling of sensitive data, such as database credentials, complies with legal and security requirements.
* Evaluate the security measures in place to protect sensitive information.

2.3.5: Schedule Feasibility:

* Assess the timeline adherence requirement, which allows for no more than a 5% deviation from the predefined timeline.
* Determine if the project can be completed within the specified timeline.
* Consider the potential challenges that could affect the project schedule.

2.3.6: Resource Feasibility:

* Review the availability of skilled resources for development and testing.
* Ensure that the team has access to necessary tools and environments.
* Assess the documentation and training resources needed for user adoption.

2.3.7: Market Feasibility:

* Evaluate the market conditions and competitive landscape to assess the potential impact of faster time-to-market on revenue.
* Consider whether the projected additional revenue of $50,000 per year is realistic based on the business's industry and customer base.

2.3.8: Scalability Feasibility:

* Confirm that the framework is designed to accommodate future growth, including up to 10,000 test cases and 500 VMs.
* Assess the framework's ability to scale with increased testing demands.

2.3.9: Usability Feasibility:

* Ensure that the documentation and training provided make the framework easy for users to adopt and use effectively.
* Consider usability testing and feedback from initial users.

2.3.10: Maintainability Feasibility:

* Assess the framework's codebase for maintainability and ease of future enhancements.
* Evaluate the process for updating documentation and ensuring it remains accurate and up to date.
* Based on the feasibility analysis, the project appears to be technically feasible and economically viable, with potential for substantial savings and efficiency gains.

**2.4: Verb/Noun Textual Analysis to determine Use-Case Actors and Methods:**

**2.4.1: Actors:**

Hub

Node

MySQL Server

Selenium Grid

**2.4.2: Methods:**

**2.4.2.1: MySQL Server:**

readTable ("Test Cases")

getValue (String tableName, String testcase, String columnName)

updateValue (String tableName, String testcase, String columnName, String value)

**2.4.2.2: Hub:**

main()

submitTask(String testCaseName, String methodName, Runnable task)

initReport()

addTestResult(String testCase, boolean passed, long time, String screenshotPath)

finalizeReport()

addTestCase(String testCaseName, long executionTime, String screenshotPath, boolean passed)

saveReport(String filePath)

getSystemCpuLoad()

getMemoryUtilization()

**2.4.2.3: Node:**

executeTestCase(String methodName, Object... args)

captureAndSaveScreenshot(String screenshotName, boolean passed)

method1(String ScreenshotName)

navigateToUrl(String url)

click(By locator)

pressEnter(By locator)

sendKeys(By locator, String text)

getText(By locator)

waitForElementToBeVisible(By locator)

waitForElementToBeClickable(By locator)

StockPriceSearch(WebDriver driver, String TestCaseName)

**2.4.2.4: Selenium Grid:**

getDriver()

quitDriver()

**2.5: Requirements Definition:**

**2.5.1: Functional Requirements:**

2.5.1.1: Test Case Management (TCM) Integration:

2.5.1.1.1: The system shall be capable of importing test case details, including test case names, dependencies, and other relevant information, from an database.

2.5.1.2: Parallel Test Execution:

2.5.1.2.1: The system shall generate separate threads for each test case to enable parallel execution.

2.5.1.2.2: Users should be able to configure the number of concurrent threads for parallel execution.

2.5.1.3: Database Interaction:

2.5.1.3.1: The system shall interact with a database to retrieve and store data required for test case execution.

2.5.1.3.2: Before test execution, the system shall retrieve necessary data from the database.

2.5.1.3.3: After test execution, the system shall store relevant data in the database.

2.5.1.4: Dynamic Test Case Distribution:

2.5.1.4.1: The system (Selenium Grid Hub) shall dynamically distribute test cases to node machines based on the current system load.

2.5.1.4.2: Test cases shall be optimally allocated to available nodes to maximize resource utilization.

2.5.1.4.3: Distribution logic shall ensure that no node receives more test cases than its capacity allows.

2.5.1.4.4: The framework shall utilize all available threads on a node machine before distributing test cases to the next node machine.

2.5.1.5: Consolidated Reporting:

2.5.1.5.1: The system shall generate a common consolidated test execution report on the hub machine.

2.5.1.5.2: The consolidated report shall include test results from all nodes, providing a unified view of test outcomes.

2.5.1.5.3: The report should be easily accessible to relevant stakeholders.

**2.5.2: Non-Functional Requirements:**

2.5.2.1: Performance:

2.5.2.1.1: The system shall aim to reduce the test execution time by a minimum of 30% compared to the previous manual distribution method.

2.5.2.1.2: The parallel execution mechanism shall achieve a system utilization rate of at least 80%.

2.5.2.1.3: The system (selenium grid hub) shall respond to system load changes swiftly and optimize test case distribution in real-time.

2.5.2.2: Scalability:

2.5.2.2.1: The framework shall be scalable to accommodate up to 10,000 test cases and up to 500 VMs.

2.5.2.3: Reliability:

2.5.2.3.1: The framework shall have an uptime of at least 99%.

2.5.2.4: Usability:

2.5.2.4.1: Documentation should be provided for setting up and using the framework effectively. Documentation shall be comprehensive, covering at least 90% of the framework's features.

2.5.2.5: Maintainability:

2.5.2.5.1: The codebase should be well-structured and maintainable to accommodate future enhancements or changes.

2.5.2.5.2: Documentation shall be updated within 2 weeks of any major feature update or bug fix.

2.5.2.6: Compatibility:

2.5.2.6.1: The framework should be compatible with different browsers and

operating system.

2.5.2.7: Security:

2.5.2.7.1: Sensitive data, such as database credentials, should be securely stored and transmitted.

2.5.2.8: Timeline Adherence:

2.5.2.8.1: The optimization project should adhere to a predefined timeline to minimize disruption to ongoing development and release cycles. The project shall be completed within a predefined timeline, with no more than a 5% deviation.

2.5.2.8.2: Changes should be phased or scheduled during low-activity periods to mitigate impact.

2.5.2.9: Reporting:

2.5.2.9.1: The framework should continue to generate detailed HTML reports within 2 minutes of test completion, that include information about test execution, including testcase names and execution times, report if there is any failure.

2.5.2.10: Resource Utilization:

2.5.2.10.1: To ensure that test cases are executed in the most resource-efficient manner, the framework should fully utilize all the threads on each node machine before moving on to the next available node. This means that if a node machine is assigned 'n' threads, all 'n' threads should be occupied with test case execution before the framework starts distributing test cases to the next node machine.

**3: Functional Model:**

**3.1: Use-Case descriptions:**The "Execute Test Case" use case, identified as UC-001 and of critical importance, involves the Node as the primary actor. This process begins when a Test Engineer starts the hub, which automatically triggers the execution of test cases on nodes. The hub generates threads for parallel test execution and distributes these to nodes via Selenium Grid. Each Node executes its assigned test case, retrieves necessary data from a MySQL database, and upon completion, returns control to the hub for report generation. This use case is integral for the efficient execution and management of automated tests in the optimized Java Selenium framework (Refer Fig1: Use-Case Description of Execute Test Cases in the provided documentation).

|  |  |  |  |
| --- | --- | --- | --- |
| Use Case Name: Execute Test Case | ID: UC-001 | | Importance Level: Critical |
| Primary Actor: Node. | | Use Case Type: Detail, Essential | |
| Stakeholders and Interests:   1. Hub: Generates the Threads that are needed to execute the testcase. 2. Node: Executes the testcase 3. MySQL Server: will give necessary data that is required to run the testcase. | | | |
| Brief Description:  The Node will Executes the TestCase with the help of thread generated by the hub, the node will fetch the test data from the MySQL table which is necessary to run the testcase, after running the testcase the desired data will be stored in the database table, and then the execution is passed to hub to generate report. | | | |
| Trigger: Test Engineer starts the hub, then hub will auto trigger the testcase to execute them on nodes.  Type: Internal | | | |
| Relationships:  Association:  The "Execute Test Case" use case is associated with the following actors and components:   1. Hub: Generates threads and distributes test cases. 2. Node: Executes test cases, fetches test data, and updates data to the database. 3. MySQL Server: Provides necessary data for test case execution. 4. Selenium Grid: Facilitates the distribution of test cases to nodes.   This use case relies on these associated actors and components to perform its functions.  Include:   1. trigger the database to fetch data that is necessary for testcase execution. 2. update the generated data to the database.   Extend:   1. TestCase Methods should be arranged in logical order. 2. Should handle the web elements properly.   Generalization: Generating Testcase Report. | | | |
| Normal Flow of Events:   1. The Hub gets the testcases and related methods from database to execute. 2. Hub will generate threads to run testcases in parallel. 3. Selenium Grid will distribute the testcases to nodes. 4. Node Machine will execute the testcases. 5. Node will send back the control to hub. | | | |
| Sub Flows:  SF1: Node will trigger the database to fetch data that is necessary for testcase execution.  SF2: Node will update the generated data to database. | | | |
| Alternate/Exceptional Flows:  EF1: If Testcases methods are not properly arranged in order testcase will fail or produce unexpected results.  EF2: Should Handel the Web elements properly by keeping proper wait statements when it is necessary, otherwise the element can’t be found by node and testcase will fail. An error is logged in the console.  EF3: If there's an issue in connecting with the MySQL server, the data cannot be updated. An error is logged in the console. | | | |

Fig1: Use-Case Description of Execute Test Cases.

**3.2: Use-Case diagram:**

The "Use-Case diagram" is revolving around three primary functionalities of a Load based Automated Test Distribution Framework: "Import Testcases", "Distribute Test Cases", and "Execute Test Cases". Each functionality involves different system actors like MySQL server, Selenium Grid, Hub, and Node. The diagram also includes a "Generate Report" use case, highlighting the interactions and dependencies among these components. For a visual representation of these interactions and the complete structure of the system's use cases, you should refer to Fig2: Use case Diagram of Load Based Automated Test Distribution Framework.

**3.2.1: Import Testcases:**

This use case involves system actor MySQL server (refer to Fig2), to import the testcases and their respective methods into the framework. The framework uses these test methods to execute the testcases.

**3.2.2: Distribute Test Cases:**

This use case involves system actor Selenium Grid (refer to Fig2), to distribute testcases to different nodes based on their respective load, Selenium Grid will read the number of threads running in a particular node in real time and decides to which node the new testcase should be distributed. If current node has any free threads, then the new testcase will be runed on current node itself otherwise if current node is full then it will be runed on new node.

**3.2.3: Execute Test Cases:**

This use case involves system actors, such as Hub and Node (refer to Fig2), Hub will generate the Threads that are necessary to run tests in parallel, then these test cases was sent to selenium grid, this will distribute the testcases to nodes based on their respective load, then the nodes will run their respective testcase in their machines and sends back the control to hub after completion of execution. This process will continue until all the testcases were executed.

**3.2.4: Generate Report:**

This use case involves system actors, such as Hub and Node (refer to Fig2), Node will execute the testcases and takes necessary screenshots and passes the control to hub, then hub will consolidate all the screenshots generated by nodes, and includes testcase names, time taken to run testcase and produces a html report. At end of the report, it also includes total no of tests passed, total no of tests failed, and total time taken to execute the testcases.

A diagram of a company

Description automatically generated with medium confidence

Fig2: Use case Diagram of Load Based Automated Test Distribution Framework.

**3.3: Activity diagram:**

I have implemented the activity diagram of whole system (refer to Fig3: Activity Diagram of Load Based Automated Test Distribution Framework) but will explain how I implemented each use case in activity diagram.

**3.3.1: Import Test Cases:**

In below Activity Diagram (refer to Fig3) this use case would depict the sequence of actions that occur when we try to import testcases from database. First, we will connect to database then we will check whether “testcases” table is present or not. If table is present, then we will fetch the data present in the table and store it in a variable. If the table is not present, then we will print error message to console and terminate the execution.

**3.3.2: Distribute the testcases:**

In below Activity Diagram (refer to Fig3) this use case would start from creating multiple threads to execute testcases in parallel. First, we will check hub’s CPU load and memory utilization if these are below certain threshold then the hub will get the current status that is number of threads running on node, if all the threads on a node are occupied then the testcase will be send to new node, if not the testcase will run on current node itself.

**3.3.3: Execute testcase:**

In below Activity Diagram (refer to Fig3) this use case would start after sending the testcase to a node. We will run the testcase in node by using data that is stored in database. After running testcase we will store the generated data in the database and then we will pass the testcase. If any errors are generated due to certain reasons like web element not present, table does not present in database etc... then we will fail the testcase and send the execution back to hub.

**3.3.4: Generate Report:**

This is the last action in the activity diagram (refer to Fig3), where after receiving all the testcases status and screenshot from respective node, the hub will generate the report based on the received status and screenshots, and hub will add some extra features to report like total number of tests passed, failed and total time taken to execute the testcases.

A blue diagram with black text

Description automatically generated

Fig3: Activity Diagram of Load Based Automated Test Distribution Framework

**4: Structural Model:**

The "Structural Model" in this documentation provides an in-depth view of the Load Based Automated Test Distribution Framework, depicted through class and object diagrams. The class diagram illustrates the system's architecture, showing classes, their attributes, methods, and interconnections. It serves to clarify how various components like BasePage, WebDriverManager, and TestRunner etc.. are structured and relate to each other. The object diagram complements this by showcasing specific instances of these classes at a particular moment, highlighting the system's static aspects at runtime. For a visual representation of this structural organization, you should refer to Fig4: Class diagram of Load Based Automated Test Distribution Framework and Fig5: Object diagram of Load Based Automated Test Distribution Framework.

**4.1: Class diagram:**

A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods) (refer to Fig4), and the relationships among the classes. The class diagram is widely used in software engineering to represent the static view of an application.

In this project, the class diagram represents the static structure of Load Based Automated Test Distribution Framework. Each class like BasePage, WebDriverManager, DynamicThreadManager, TestRunner, ReportManager, StockPrice, TestCaseExecutor, TestMethods, DatabaseReader and SystemInfoUtil, represents an entity in my system (refer to Fig4). The attributes within each class represent the data associated with that entity. The methods in each class represent the operations or behaviour associated with that entity. The relationships between classes, represent how these entities interact with each other.

A screenshot of a computer

Description automatically generated

Fig4: Class diagram of Load Based Automated Test Distribution Framework.

**4.2: Object diagram:**

An object diagram is used to model the static view of a system at a specific moment in time. It shows a complete or partial view of the structure of a modelled system at a specific time. Object diagrams are derived from class diagrams (Refer Fig4: Class diagram of Load Based Automated Test Distribution Framework), so they are more specific and instantiated versions of class diagrams.

In my project, the object diagrams diagrams (Refer Fig5: Object diagram of Load Based Automated Test Distribution Framework) are specific instances of the classes. For instance, the runner object is a specific instance of the TestRunner class with particular values for its attributes. The same goes for objects of BasePage, WebDriverManager, DynamicThreadManager, ReportManager, StockPrice, TestCaseExecutor, TestMethods, DatabaseReader and SystemInfoUtil.

A computer screen shot of a computer code

Description automatically generated with medium confidence

Fig5: Object diagram of Load Based Automated Test Distribution Framework

**5: Behavioural Model:**

A behavioural model in software engineering is a representation of the interaction between objects in the system and how these interactions change the state of these objects over time. These interactions typically occur through the calling of methods and functions, which affect the attributes of these objects.

The behavioural model for each of the use cases is explained below.

* Import Testcases:

This use case involves system actor MySQL server, which imports the testcases table from the database, framework uses this data to run the testcases by running the methods of the respective testcase.

* Distribute Test Cases:

This use case involves system actor Selenium Grid, this will hit all the nodes and get the status of these nodes and selects any one node to run the current testcase. To run the testcase on that node, selenium grid will get the driver instance from that respective system and assigns to the driver instance in the hub machine. Then when we use the driver to run the testcase, the testcase will run on the node machine to which driver is linked.

* Execute Test Cases:

This use case involves system actors, such as Hub and Node, hub will generate the threads that are necessary to run testcases in parallel. For every testcase hub will check its CPU load and memory utilization if it below certain threshold then it will send the testcases to selenium grid, then selenium grid will distribute to nodes based on there current load, after that node will run the testcase by using methods in basepage class, and also it will access the database to get the data that is necessary to run the test. After completion of test, it stores the necessary data in database, node also captures the screenshot of the webpage to know until where the testcase is executed, if all the lines of testcase are executed then the node will return true, otherwise it will return false status to hub.

* Generate Report:

This use case involves system actors, such as Hub and Node, Node will execute the testcases and takes necessary screenshots and passes the control to hub with the status of the testcase whether it was passed or failed, in this way hub will get status of every node and stores it in map of string and test result, where string will store testcase name, test result object will store time, status and screenshot path of every testcase. After completion of execution by all the nodes, hub will prepare the html report based on the stored data in map object. Where it will display testcase name, time taken to execute the testcase, whether the test is passed or failed, and screenshot of last webpage reached by that testcase. Finally, it also displays total number of testcases passed, failed and time taken to execute these testcases.

Note: I have combined all behavioural model of use cases and drawn in a single sequence diagram. So, that we can understand all the operations clearly what is happening on an entire system level.

**5.1: Sequence diagram:**

The Sequence diagram, (Refer Fig6: Sequence diagram of Load Based Automated Test Distribution Framework), effectively illustrates the interactions between objects in your Java Selenium Framework over time. This diagram showcases the dynamic behaviour of the system, particularly focusing on the sequence of messages exchanged between various objects during the execution of test cases. It includes interactions among key components like the Test Runner, Selenium Grid, Nodes, and Database, clearly depicting how these elements collaborate to execute and manage test cases.

Each object in the diagram is represented by a lifeline, and the messages or method calls between these objects are shown as arrows. The diagram starts with the initiation of the test process, likely from a Test Runner, and proceeds through various steps such as fetching test data from the database, distributing test cases to different nodes via the Selenium Grid, and executing these test cases. The nodes then return their execution results back to the Test Runner, which ultimately compiles these results into a report. This sequential representation is crucial for understanding the flow of operations within the testing framework, highlighting the interaction patterns that are vital for efficient and accurate test execution.

A diagram of a project

Description automatically generatedFig6: Sequence diagram of Load Based Automated Test Distribution Framework

**5.2: Words connecting the structural and behavioural models on how I generated them from my functional models:**

First, I wrote down main functionality of my framework in terms of end user. These main functionalities are drawn as use case diagram (Refer Fig2). Then I extended these functionaries in a logical way, where I wrote some conditions or operations that are needed to perform all the functionalise and structured them in a logical way which is my activity diagram (Refer Fig3).

5.2.1: Class diagram from use case diagram and activity diagram:

For every functionality in use case diagram (Refer Fig2), I tried to replicate them in terms of class structure. I noted down all the sub functionalities that are need, to implement a use case, for example to implement “import testcases” use case I need to create a class where it can do certain operations like “read table” where it can read all the rows present in a table and store them in array of strings. I also need some variable where I can store database access URL with database name, database logins such as username and password. And also, to implement “execute testcase” use case, I need a class to declare a driver instance, I need a class to invoke the test methods using the testcases data that we got from database, I also need some extra methods to interact with database to get the data and to update the data into database. In the similar way I noted down all the classes, variable and methos need in every class to perform sub functionalities.

Then I draw relationships between these classes based on the communication between methods and how this will impact objects state.

5.2.2: Object diagram from use case diagram and activity diagram:

After generating class diagram from use case and activity diagram. Based on the variable’s values at a particular instance I drawn object diagram (Refer Fig5). If a class does not have class level variables, then I did not represent those class object’s because there will be no change in state of object from time to time as there are no instance variables in it.

5.2.3: Sequence diagram from use case diagram and activity diagram:

I divided use case diagram into small functionalities, where for every functionality we need some classes, these classes and their relationship is shown in class diagram. from class diagram based on static view I derived object diagram, for a particular use case I noted down what are all the classes need, then by using activity diagram logic and object diagram methods, relationships. I came to a conclusion of sequential calls between methods that are performed to complete a particular functionality. In the similar way for all the use cases, I have noted down the sequential method calls that ae need to perform the functionality of that use case. Then I connect all the use cases to get the sequential diagram of entire framework or system. Let us take the example of “import testcases” use case, for this use case we have database reader class and, in this class, we have “readtable” method so in sequential diagram we called MySQL database from “test runner” class by using “readtable” method. This methods will connect with database reader class which reads the table and sends data to class level variable which is in test runner class. In the same ways object interactions by using methods are shown in sequence diagram (Refer Fig6).

**6: Design Model:**

The Design Model, as detailed in (Fig7 and Fig8), encapsulates the structural arrangement of the Java Selenium Framework. It comprises package diagrams that provide a clear visualization of the framework's modular design, showing how different classes and components are organized and interact within distinct packages. These diagrams offer a comprehensive view of the framework's architecture, illustrating the relationships and dependencies between various entities. The first diagram (Fig7) presents an overview of the framework with classes categorized within their respective packages, while the second diagram (Fig8) delves deeper, highlighting the classes and their interrelationships within each package. This model is instrumental in understanding the framework's overall structure, aiding in its maintenance and scalability.

**6.1: Package diagram:**

The package diagram in this project includes two main diagrams. The first one (Refer Fig7) displays the overall Load Based Automated Test Distribution Framework with classes organized within their respective packages. The second diagram (Refer Fig8) provides a more comprehensive view of the same framework, highlighting the classes and their interrelationships within each package. These diagrams are crucial for understanding the modular structure of this system and how different components are grouped and interact​​.

Several blue folders with text

Description automatically generated

Fig7: Package diagram of Load Based Automated Test Distribution Framework with classes in respective package.

A diagram of a computer

Description automatically generated

Fig8: Package diagram of Load Based Automated Test Distribution Framework.

**6.2: Contracts:**

|  |  |  |
| --- | --- | --- |
| Method Name: stockPriceSearch | Class Name: StockPrice | ID: C-001 |
| Clients (Consumers): TestMethods class (specifically the method1 method) | | |
| Associated Use Cases: Retrieving and storing stock price data for test cases | | |
| Description of Responsibilities:  Navigate to the Google home page, search for a specified stock's price, capture the current price details from the search results, and update the values in the test data database. | | |
| Arguments Received:  WebDriver driver - the web driver instance for browser interaction.  String testCaseName - the name of the test case to retrieve and update the stock name | | |
| Type of Value Returned:  boolean - true if the method completes successfully, false if an exception is encountered. | | |
| Pre-Conditions:  - A valid WebDriver instance must be provided.  - Internet connectivity must be present.  - The specified stock name must exist in the TestData database table under the given test case name. | | |
| Post-Conditions:  - The stock price data is updated in the TestData database table.  - A boolean value indicating the success or failure of the method is returned. | | |

Fig9: Contract for the stockPriceSearch Method of the StockPrice Class

**6.3: Method Specifications:**

|  |  |  |
| --- | --- | --- |
| Method Name: stockPriceSearch() | Class Name: StockPrice | ID: MS-001 |
| Contract ID: C-001 | Programmer: Alex | Date Due: 12/31/2023 |
| Programming Language:  Selenium WebDriver Java SQL | | |
| Triggers/Events:  The method is called when stock price information retrieval and update are required for a test case. | | |

|  |  |
| --- | --- |
| Arguments Received:  Data Type: | Notes: |
| WebDriver String | The web driver instance to interact with the browser.  The name of the test case to be used to retrieve and update stock data.  The WebDriver instance and testCaseName are required to perform the search and update actions. |

|  |  |  |
| --- | --- | --- |
| Messages Sent & Arguments Passed:  ClassName.MethodName: | Argument  Data Type: | Notes: |
| stockprice.stockPriceSearch(driver, testcaseName); | WebDriver driver, String testCaseName | Driver instance is passed to maintain separate instance for every testcase when executing in parallel, testcaseName was passed to get the values and to update value in a particular testcase related row in database. |

|  |  |
| --- | --- |
| Argument Returned:  Data Type: | Notes: |
| Boolean | True if the stock name was retrieved from database and search for stock price in the browser and store the price in database. |
| Algorithm Specification:   * Navigate to the Google search page using the provided WebDriver instance. * Input the search query for the stock price into the search box and submit the search. * Parse the search results to find stock price information. * Call the DatabaseReader.updateValue method to update the stock price in the test data database. * Return true if the operation is successful, or false if any exceptions are caught during the process. | |
| Misc.Notes:   * The method assumes the presence of a stable internet connection and correct configuration of the WebDriver instance. The stock symbol or name used in the search query must be valid and present in the database under the given test case name. * The method runs within the test automation framework and will be executed as part of automated test cases. | |

Fig10: Method Specification of stockPriceSearch Method.

**7: Data Management Layer Design:**

The Data Management Layer Design in this project, as depicted in Fig11 and Fig12, focuses on how data is structured and managed within the Java Selenium Framework. This design includes a detailed database schema and an object persistence format. The database schema (refer Fig12) outlines the structure of the database tables, such as 'TestCases' and 'TestData', including their relationships and key attributes. It ensures efficient storage and retrieval of test-related data. The object persistence format (refer Fig11) maps object attributes to database columns, maintaining data integrity and facilitating smooth data transactions between the application and the database. This layer is crucial for the effective management of test data, ensuring consistency and accuracy in test executions.

**7.1: Object persistence format:**

The Object persistence format in this document is a table that maps object attributes to database columns (refer Fig11). It includes attributes such as **testCaseID**, **testCaseName**, **stockName**, **openPrice**, **highPrice**, and **lowPrice**, with corresponding data types and database column types. This format is integral for understanding how data is transferred and stored in the database, ensuring consistency and integrity of data management​​.

|  |  |  |  |
| --- | --- | --- | --- |
| **Object Attribute** | **Attribute Data Type** | **Database Column** | **Database Column Type** |
| testCaseID | int | TestCaseID | INT |
| testCaseName | String | TestCaseName | VARCHAR |
| stockName | String | stockname | VARCHAR |
| openPrice | double | OpenPrice | DECIMAL |
| highPrice | double | HighPrice | DECIMAL |
| lowPrice | double | LowPrice | DECIMAL |

Fig11:Object persistence format table that maps object attributes to database columns.

**7.2: Database schema:**

The database schema for the "Optimizing Java Selenium Framework with Automated Test Distribution and Full Resource Utilization" project includes two main tables - TestCases and TestData (Refer Fig12). These tables are crucial for storing and managing test-related data and their execution outcomes, as visualized in the provided diagram (Refer Fig12).

**7.2.1: TestCases Table:** This table is foundational for keeping a record of each test case within the system. It has three columns:

* + TestCaseId: Serves as the PRIMARY KEY, a unique identifier for each test case entry.
  + TestCaseName: A VARCHAR(255) field that stores the name of the test case, marked as NOT NULL and UNIQUE, indicating that each test case name must be distinct and present.
  + AssociatedMethod: A VARCHAR(255) field that indicates the method associated with the test case, ensuring that the method name is always provided (NOT NULL).

**7.2.2: TestData Table:** This table contains the data used during the test execution. It has five columns:

* + TestDataId: Identified as the PRIMARY KEY, it uniquely identifies each data record.
  + TestCaseName: A VARCHAR(255) field, marked as NOT NULL FOREIGN KEY, which references the TestCaseName in the TestCases table, establishing a relationship between the test data and the test case it belongs to.
  + stockName: A VARCHAR(255) field, NOT NULL, which stores the name of the stock associated with the test data.
  + OpenPrice, HighPrice, LowPrice: These fields are of the DECIMAL(10, 2) type, storing the stock prices relevant to the test case execution, ensuring precision for financial calculations.

The database schema is designed with referential integrity in mind. The dashed line connecting the TestCaseName in both tables suggests a one-to-one relationship (Refer Fig12), where one test case can have one associated data entries. This setup supports the test framework's need to execute test cases based on various data inputs and record their results accordingly. The use of FOREIGN KEY constraint on the TestCaseName in the TestData table ensures data consistency, meaning every test data entry is linked to an existing test case.

**A close-up of a computer code

Description automatically generated**

Fig12:database schema for the Java Selenium Framework

**8: Human-Computer Interaction Layer Design:**

**8.1: Wireframe design of Test Report:**

The wireframe presents a structured view of the Test Execution Report for the "Optimizing Java Selenium Framework" project (Refer Fig13). The report is delineated into several key sections, visually represented with clarity and emphasis on the outcomes of the test cases.

**8.1.1: Title Section:**

* The report features a prominent title, "Test Execution Report," which is followed by a date and time stamp formatted as "YYYY-MM-DD HH:MM:SS" (Refer Fig13). This indicates when the test execution was completed, providing a temporal context for the report.

**8.1.2: Test Case Results Table:**

* The main body of the report is a table that lists all executed test cases (Refer Fig13).
* Each row in the table corresponds to an individual test case, with columns for "Test Case", "Execution Time (Seconds)", and "Screenshot".
* The "Test Case" column identifies the test case by its name.
* The "Execution Time (Seconds)" column displays the time taken to execute each test case.
* The "Screenshot" column presumably would link to or display a thumbnail of the screenshot captured during the test case execution. It serves as evidence and a point of reference for reviewing the test case's execution state.
* The color coding within the table provides immediate visual feedback on the status of each test case: green indicates a passed test case, and red indicates a failed one.

**8.1.3: Summary Section:**

* Below the table, there is a "Summary" section that encapsulates the aggregate results of the test executions (Refer Fig13).
* It includes the total number of test cases passed and failed, providing a quick overview of the testing outcomes.
* Additionally, it reports the "Total Execution Time," summing up the time taken for all test cases, which allows for a quick assessment of the overall time efficiency.

**8.1.4: Actual Execution Time:**

* At the bottom, there is a separate box labeled "Actual Execution Time," which shows the "Total Actual Execution Time" (Refer Fig13). This metric is essential for understanding the performance of the test suite as it indicates the wall-clock time from the start to the end of the entire test suite's execution.

The design of the report in the wireframe is utilitarian, focusing on delivering essential information in an easily digestible format. By using color coding and clear sectioning, it allows test engineers and stakeholders to quickly assess the health and performance of the application under test. The inclusion of timestamps and execution times helps in tracking performance over time and in identifying any outliers or trends in test execution durations.

**A screenshot of a report

Description automatically generated**

Fig13: wireframe diagram of Test Execution Report for the Optimizing Java Selenium Framework

**9: Physical Architecture Layer Design:**

**9.1: Hardware and Software Specification Overview:**

For the design of Java Selenium Framework with Automated Test Distribution and Full Resource Utilization, we just need a client where our program runs and a database server to store testcases related data as shown in Fig14: Summary Table of Hardware and Software Specifications.

9.1.1: Standard Client Specifications:

* Operating System: The client machine is specified to run on Windows 10, optimized with Chrome Browser 90 for the best web automation experience (refer Fig14).
* Special Software: The Java Runtime Environment (Java SE 17) is installed to support Java-based automation scripts. Selenium Grid 4.0.0 is also included to allow for distributed test execution and cross-browser testing directly from the client machine (refer Fig14).
* Hardware: A robust setup is provided for the client with 16 GB of Memory and a 512 GB SSD, ensuring swift performance during test executions (refer Fig14). The system is powered by an Intel Core i7 processor or a comparable alternative, which is capable of handling the automation workload efficiently.
* Network: The client is equipped to connect over a 100 Mbps Ethernet or use High-speed Wireless networking to ensure seamless communication and data transfer during test automation processes (refer Fig14).

9.1.2: Standard Database Server Specifications:

* Operating System: The database server runs on Linux Ubuntu 20.04 LTS (refer Fig14), which is known for its stability and security, making it a preferred choice for database operations.
* Special Software: MySQL Server 8.0 is the selected database management system (refer Fig14), which provides the necessary features and performance for managing the test data effectively.
* Hardware: The server specifications are designed to handle large volumes of data with 16 GB Memory and storage options of 1-2 TB HDD (refer Fig14). It also includes an Intel Xeon processor or similar, capable of supporting the demands of a database server in a test automation environment.
* Network: A standard 100 Mbps Ethernet network connection is specified for the database server to facilitate robust and fast data handling capabilities required during automated test executions (refer Fig14).

**Summary Table:**

| **Specification** | **Standard Client** | **Standard Database Server** |
| --- | --- | --- |
| Operating System | Windows 10 / Chrome Browser 90 | Linux Ubuntu 20.04 LTS |
| Special Software | Java Runtime Environment (Java SE 17) Selenium Grid 4.0.0 | MySQL Server 8.0 |
| Hardware | 16 GB Memory / 512 GB SSD / Intel Core i7 or similar | 16 GB Memory / 1-2 TB HDD / Intel Xeon or similar |
| Network | 100 Mbps Ethernet or High-speed Wireless | 100 Mbps Ethernet |

Fig14: Summary Table of Hardware and Software Specifications

**9.2: Operational Requirements:**

**9.2.1: Framework Efficiency**: Achieve a reduction in average test execution times by at least 30% compared to the current non-optimized baseline.

**9.2.2: Resource Utilization**: Ensure CPU and memory utilization is maximized to at least 70% of capacity during testing to prevent underutilization while leaving a buffer for system stability.

**9.2.3: Scalability**: The framework should be able to handle a minimum of a 50% increase in the number of test cases without degradation in performance.

**9.2.4: Parallel Execution**: The system should support parallel execution of test cases to maximize test throughput and reduce overall test execution time.

**9.2.5: Database Integration**: Test data retrieval and storage operations should take no longer than 2 seconds per operation.

**9.2.6: Test Case Management**: Capability to read and write 1000 test cases from and to the database within 5 minutes.

**9.2.7: Load Balancing**: The system should distribute test cases across available nodes effectively based on system load.

**9.2.8: Reporting**: Generate and display comprehensive test execution reports within 60 seconds of completing the test suite.

**9.2.9: Reliability**: Achieve 99% system uptime, excluding scheduled maintenance windows.

**9.2.10: Usability**: The interface should be intuitive enough that a new user can initiate test runs and monitor progress with no more than 30 minutes of initial training.

**9.2.11: Maintenance**: The system should be easy to maintain and update, with clear documentation for any updates or changes to the testing framework.

**9.2.12: Continuous Integration**: The framework should integrate with existing CI/CD pipelines and support automated triggering of tests within 5 minutes of a new build deployment.

**9.2.13: Disaster Recovery**: Implement a disaster recovery plan that allows for full system recovery within 4 hours following any failure.

**9.2.14: Training**: training materials should cover at least 90% of the framework functionalities.

**9.3: Performance Requirements:**

**9.3.1: Test Execution Time Reduction**:

Reduce the average test execution time by at least 30% compared to the current baseline without the optimized framework.

**9.3.2: System Resource Utilization**:

9.3.2.1: CPU Utilization: The framework should not exceed 80% average CPU utilization on any application server during test execution.

9.3.2.2: Memory Utilization: Maintain at least 20% free memory capacity during peak testing periods.

**9.3.3: Parallel Execution Capacity**:

The system should support the simultaneous execution of a minimum of X number of test cases per node, where X should be determined based on system specifications and load testing results.

**9.3.4: Database Transaction Times**:

9.3.4.1: Read Operations: Average database read operations should not exceed 100 milliseconds.

9.3.4.2: Write Operations: Average database write operations should not exceed 150 milliseconds.

**9.3.5: Load Distribution**:

Test case distribution across nodes should not take more than 5 seconds from the point of triggering the test execution.

**9.3.6: Report Generation**:

Consolidated test execution reports should be generated within 60 seconds of completing the test suite.

**9.3.7: Scalability**:

The system must scale to handle a 50% increase in test cases without a corresponding increase in average test execution time.

**9.3.8: Reliability**:

9.3.8.1: Uptime: The testing framework should achieve an uptime of 99% outside of scheduled maintenance windows.

9.3.8.2: Error Rate: The system should maintain an error rate below 1% for test execution failures due to system errors (not including test case errors).

**9.3.9: Browser and Device Coverage**:

The framework should support and be tested across at least 95% of the target browser and device combinations.

**9.3.10: Test Data Management**:

Test data retrieval and storage operations should not introduce more than a 10% overhead to the overall test execution time.

**9.3.11: Network Bandwidth**:

The system should be tested to support a minimum bandwidth of 50 Mbps to ensure that network constraints do not impact test execution performance.

**10: Construction:**

**10.1: Unit Tests:**

**Test Plan: BlackBox Testing**

**Program ID:** Selenium WebDriver Testing Framework **Version Number:** 1.0

**Tester:** Alex  **Date Designed:** 11/10/2023 **Date Conducted:** 11/20/2023

**Results:** Passed **Open items:** None

**Test ID:** SDW-001 **Requirement addressed:** Verify URL contains the correct stock

name after search

**Objective:** To verify that the **verifyUrlContainsStockName** method of the **BlackBoxTestCase** class correctly identifies if the current URL of the web page contains the specified stock name.

**Test cases:**

|  |  |  |
| --- | --- | --- |
| **Testcase ID** | **Data Field** | **Value Entered** |
| 1.SDW-001 | Stock Name | "Apple" |
| 2.SDW-002 | Stock Name | "Google" |
| 3.SDW-003 | Stock Name | "InvalidName" |

**Script:**

1. Navigate to a stock-related page using WebDriver and call **verifyUrlContainsStockName** with "Apple" (Test Case SDW-001).
2. Navigate to a different stock-related page and call the same method with "Google" (Test Case SDW-002).
3. Call the method with an invalid stock name, such as "InvalidName" (Test Case SDW-003).

**Expected Results/Notes:**

1. The method should return **true** for valid stock names present in the URL.
2. It should return **false** for invalid or non-existent stock names.

**Actual Results/Notes:**

1. **Test Case SDW-001 (Stock Name: "Apple")**: The method **verifyUrlContainsStockName** correctly identified the presence of "Apple" in the URL. Test Case SDW-001 passed.
2. **Test Case SDW-002 (Stock Name: "Google")**: The method **verifyUrlContainsStockName** correctly identified the presence of "Google" in the URL. Test Case SDW-002 passed.
3. **Test Case SDW-003 (Stock Name: "InvalidName")**: The method **verifyUrlContainsStockName** correctly returned **false**, indicating the stock name "InvalidName" was not in the URL. Test Case SDW-003 passed.

**Test Plan: Whitebox Testing**

**Program ID:** Selenium WebDriver Testing Framework **Version Number:** 1.0

**Tester:** Alex  **Date Designed:** 11/12/2023 **Date Conducted:** 11/20/2023

**Results:** Passed **Open items:** None

**Test ID:** SDW-002 **Requirement addressed:** Validate correct reading of database tables

**Objective**: To verify that the **validateDatabaseReader** method of the **WhiteBoxTestCase** class accurately validates the number of rows read from a database table.

**Test cases:**

|  |  |  |
| --- | --- | --- |
| **Testcase ID** | **Data Field** | **Value Entered** |
| 1.DB-001 | Table Name | "TestCases" |
| 2.DB-002 | Table Name | "InvalidTable" |

**Script:**

1. Call **validateDatabaseReader** with "TestCases" and the expected number of rows (Test Case DB-001).
2. Call the same method with a non-existent table name "InvalidTable" (Test Case DB-002).

**Expected Results/Notes:**

1. The method should return **true** if the actual number of rows matches the expected number.
2. It should return **false** for non-existent tables or incorrect row counts.

**Actual Results/Notes:**

1. **Test Case DB-001 (Table Name: "TestCases")**: The **validateDatabaseReader** method accurately validated the number of rows in the "TestCases" table. Test Case DB-001 passed.
2. **Test Case DB-002 (Table Name: "InvalidTable")**: The **validateDatabaseReader** method correctly returned **false** as the table "InvalidTable" does not exist. Test Case DB-002 passed.

**10.2: System Requirements Testing:**

Test plan for the functional requirement 2.5.1.3, which specifies that the system shall interact with a database to retrieve and store data related to testcase.

**Test Plan: Business Requirement Testing**

**Program ID:** Selenium WebDriver Testing Framework **Version Number:** 1.0

**Tester:** Alex  **Date Designed**: 11/14/2023 **Date Conducted**: 11/22/2023

**Results:** Passed **Open items:** None

**Test ID**: SDW-003 R**equirement addressed:** Validate data in the database

**Objective**: To verify that the **validateDataInDatabase** method of the **BusinessRequirementTestCase** class correctly validates specific values in the database.

**Test cases:**

|  |  |  |
| --- | --- | --- |
| **Testcase ID** | **Data Field** | **Value Entered** |
| 1.DBV-001 | Test Case Name | "TestCase1", "OpenPrice" |

**Script:**

1. Call **validateDataInDatabase** with expected values from the "TestData" table and check for specific test case details (Test Case DBV-001).

**Expected Results/Notes:**

1. The method should return **true** if the actual database value matches the expected value.

**Actual Results/Notes:**

1. **Test Case DBV-001 (Test Case Name: "TestCase1", Column Name: "OpenPrice")**: The **validateDataInDatabase** method successfully verified that the value in the "TestData" table for "TestCase1" and "OpenPrice" matched the expected value. Test Case DBV-001 passed.

**10.3: System Performance Testing:**

Test plan for the non-functional requirement 2.5.2.1, which specifies that the system shall aim to reduce the test execution time by a minimum of 30% compared to the previous manual distribution method.

**Test Plan: Performance Requirement Testing**

**Program ID:** Selenium WebDriver Testing Framework **Version Number:** 1.0

**Tester:** Alex  **Date Designed**: 11/14/2023 **Date Conducted**: 11/22/2023

**Results:** Passed **Open items:** None

**Test ID**: SDW-004 R**equirement addressed:** Test case execution time

**Objective**: To validate that the execution time of a specified test case does not exceed the expected time using **validateTestcaseExecutionTime** from **PerformanceTestCase**.

**Test cases:**

|  |  |  |
| --- | --- | --- |
| **Testcase ID** | **Data Field** | **Value Entered** |
| 1.PT-001 | Test Case Name | "TestCase2" |
| 2.PT-002 | Execution Time (ms) | 40000 |

**Script:**

1. Execute the test case "TestCase2" and validate if the execution time stays within the specified limit (Test Case PT-001).

**Expected Results/Notes:**

1. The method should return **true** if the execution time is within the expected limit.

**Actual Results/Notes:**

1. **Test Case PT-001 (Test Case Name: "TestCase2", Execution Time: 40000 ms)**: The **validateTestcaseExecutionTime** method confirmed that the execution time for "TestCase2" was within the specified limit of 40,000 milliseconds. Test Case PT-001 passed.

**11: Acknowledgement:**

I would like to express my profound gratitude to all those who have been instrumental in the realization of this project. First and foremost, my deepest appreciation goes to John Doe, VP of Quality Assurance, whose expertise, guidance, and support from the initial to the final level enabled me to develop an understanding of the subject.

My heartfelt thanks to my mentor Venkat, whose encouragement, supervision, and support from the preliminary to the concluding level enabled me to comprehend the complexities of the project. Their constant feedback and enthusiastic engagement in every step of the process have been invaluable.

A special thanks to my colleagues and the team at WeThink Technology Solutions for their insights, assistance, and constant encouragement throughout this challenging project. Their diverse perspectives and expertise have been crucial in the development of this framework.

My sincere appreciation also goes to my peers and the technical staff who provided me with the necessary resources and environment conducive for my project. Their technical insights and assistance were crucial in overcoming the challenges faced during the development of this project.

**12: Conclusion:**

This project, "Optimizing Java Selenium Framework with Automated Test Distribution and Full Resource Utilization," embarked on the challenging yet fulfilling journey of enhancing the efficiency and reliability of automated testing processes. The project successfully addressed critical issues like underutilization of system resources, manual test case distribution, lack of scalability, data management challenges, and inefficient reporting in the existing Selenium Java framework.

Through diligent work and a focused approach, we achieved significant improvements in resource utilization, test execution time reduction by at least 30%, and enhanced scalability to meet future testing needs. The introduction of dynamic thread allocation, parallel execution, database interaction, and load-based test case distribution are milestones that mark the project's success.

Moreover, the project delivered a consolidated test execution report, providing a unified view of test outcomes, which is pivotal for stakeholder assessment and decision-making. The value of this project is evident in its contribution to accelerating the speed to market, optimizing resource allocation, and reducing costs, thereby strengthening our competitive edge in the market.

In conclusion, the project not only achieved its set objectives but also laid down a robust foundation for future enhancements and scalability. It stands as a testament to the collaborative effort and technical prowess of the team and the invaluable support from all the stakeholders involved.

**13: References:**

Arya, K. V., & Verma, H. (2014, December). Keyword driven automated testing framework for web application. In 2014 9th International Conference on Industrial and Information Systems (ICIIS) (pp. 1-6). IEEE.

Garcia, B. (2022). Hands-On Selenium WebDriver with Java. O'Reilly Media, Inc.

Gundecha, U., & Avasarala, S. (2018). Selenium webdriver 3 practical guide: End-to-end automation testing for web and mobile browsers with selenium webdriver. Packt Publishing Ltd.

Krishna, R. (September 23, 2023). Selenium Grid Tutorial: Setup a Hub and Node. Guru99.

https://www.guru99.com/introduction-to-selenium-grid.html

Kumar, A., & Saxena, S. (2015, May). Data driven testing framework using selenium WebDriver. International Journal of Computer Applications, 118(18).

Leotta, M., García, B., Ricca, F., & Whitehead, J. (2023, April). Challenges of end-to-end testing with selenium WebDriver and how to face them: A survey. In 2023 IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 339-350). IEEE.

Neha, V. (February 8, 2023). Data Driven Framework in Selenium. Browserstack.

https://www.browserstack.com/guide/data-driven-framework-in-selenium

Neha, V., & Pawan, K. (August 26, 2023). Keyword Driven Framework for Selenium. Browserstack.

https://www.browserstack.com/guide/keyword-driven-framework-in-selenium