Test plan for Currency rates application

**Key tester: Katsiaryna Makarava**

**Key developer: Ivan Pavlov**

# 1. Introduction

## Objective:

The purpose of this test plan is to establish a structured approach for the creation, implementation, and maintenance of API and Web UI test automation. This includes defining the strategy, scope, and tools required to ensure robust and reliable test coverage for currency rate APIs and Web UI. The objective is to facilitate efficient testing processes, enable early detection of defects, and provide reusable automation scripts to support ongoing development and future scalability.

## Scope

The Scope section defines the features and functionalities that will be covered in the current phase of test automation. This includes the critical aspects of API testing for currency rate retrieval from 3 banks and the planned Web UI functionality. By clearly identifying the scope, this section ensures alignment with project goals and prioritization of automation activities.

|  |  |  |  |
| --- | --- | --- | --- |
| Feature/Aspect | Description | Automation Type | Priority |
| Check supported currencies | Returns list of supported currencies. | API Automation | High |
| Check if a currency is supported | Returns the response stating if the specified currency is supported. | API Automation | High |
| Check Currency rates in supported banks | Fetches currency rates from the European Central Bank, Australian Reserve Bank and Bank of Canada. | API Automation | High |
| Data Validation | Ensures the response payload matches expected structures and data formats. | API Automation | High |
| MongoDB Data Upload Verification | Verifies that fetched API data is correctly uploaded to the MongoDB database. | API Automation | High |
| MongoDB Data Retrieval Verification | Ensures data is retrieved from MongoDB instead of external API when the same endpoint is invoked again. | API Automation | High |
| API Requests Logs Verification | Confirms that logs of API requests are stored correctly in another MongoDB instance. | API Automation | Medium |
| Error Handling | Verifies proper error codes and messages for invalid requests. | API Automation | Medium |
| Web UI Automation | Tests interactions with the user interface for displaying and managing currency rate data in Google Chrome. | Web UI Automation | Medium |
| Exploratory testing | Explore endpoint behavior before and during the creation of automated tests to uncover bugs early and investigate bugs if found. | Manual (Exploratory) | High |
| Regression Testing | Automates regression tests for APIs and Web UI as new features are added. | API and Web UI Automation | High |

## Out-of-Scope

The Out-of-Scope section identifies functionalities and aspects that will not be included in the current phase of test automation. This is to manage expectations, avoid scope creep, and focus the team’s efforts on high-priority tasks. Items in this section may be revisited in future phases as the application evolves.

|  |  |  |
| --- | --- | --- |
| Feature | Reason for Exclusion | Revisit Timeline |
| Performance Testing | Performance testing requires specialized tools and expertise, not currently planned. | Later phases, if needed |
| Manual Testing | Focus is on automation, and manual testing is handled outside the scope of this plan. | N/A |
| Cross-Browser Testing | Testing the application's compatibility across multiple browsers is not included at this stage. Only Google Chrome (latest build) is in scope, as it holds the highest market share among browsers. | Phase 2 |
| Mobile Testing for Web UI | Mobile responsiveness testing is postponed until after desktop Web UI is automated. | Phase 2 |
| Security Testing | Requires additional tools and expertise beyond the current team’s focus. | Later phases, if needed |

**Note:** As of December 2024, Google Chrome holds approximately 68.34% of the global browser market share, making it the most widely used browser.

# 2. Test Automation Goals

The test automation aims to achieve the following objectives:

1. **Reduce Manual Testing Effort**  
   Minimize repetitive manual testing tasks by automating test cases, allowing the team to focus on high-value testing activities.
2. **Ensure API Functionality**  
   Validate the functionality of API endpoints for the European Central Bank (ECB), Australian Reserve Bank (ARB), and Bank of Canada (BoC), ensuring accurate currency rate retrieval and consistent behavior across the priority scenarios.
3. **Support Web UI Testing**  
   Develop and maintain automation scripts for the Web UI to ensure seamless integration with backend services and a smooth user experience.
4. **Deliver Reusable and Scalable Test Scripts**  
   Build an automation framework with reusable components that can be easily extended to accommodate future features and scaled for broader testing needs.

# 3. Team Roles and Responsibilities

This section describes the roles and responsibilities of the team members involved in the test automation initiative. Clear task distribution is aimed at ensuring efficient collaboration and accountability for achieving the project goals.

|  |  |  |
| --- | --- | --- |
| Role | Responsibilities | Tools/Resources Involved |
| Key Tester | Design, write, and execute test automation scripts for APIs and Web UI. | - IDEs: IntelliJ (for test code), VS Code (for application code) - Test automation tools: Junit5, RestAssured, Selenium, Testcontainers - .Net for application code; Java for test code |
| Write and maintain test deliverables, including the test plan, bug reports, and test results report. | - Document management tools: MS Word, Google Docs, Maven surefire report plugin |
| Document application requirements to ensure clarity and alignment between development and testing efforts. | - Document management tools: MS Word, GitHub, Notepad, Intellij IDEA |
| Validate test data and monitor test execution within containerized environments. | - Testcontainers library, Docker, Junit5 |
| Collaborate with the developer to address testing challenges and blockers. | - Telegram, GitHub |
| Key Developer | Share source code to review APIs. | GitHub |
| Assist in resolving test automation blockers (e.g., containers not starting, API stability, data consistency). | - IDEs: IntelliJ (for test code), VS Code and Rider (for app code) - .Net for application code; Java for test code - Docker - Other development tools and frameworks |
| Ensure the application is compatible with containerized testing environments. | - Docker, Testcontainers |

# 5. Test Automation Tools and Frameworks

This section outlines the tools and frameworks that will be used to develop and execute the test automation suite for the application. The team will use industry-standard tools to perform API and Web UI testing, as well as container management. Given the tech stack and the requirements of the application, the selected tools will help ensure efficient testing and maintainability.

|  |  |  |
| --- | --- | --- |
| Tool/Framework | Purpose | Usage |
| IntelliJ IDEA | Java IDE used for editing and executing test cases. | Key tester uses it for editing test cases. Developer views test code. |
| VS Code | Lightweight code editor for viewing application code. | Key tester uses it to view application code. |
| Postman | API testing tool for exploratory testing before automating test cases. | Used for manual testing and exploring API responses. |
| MongoDB Compass | GUI tool for interacting with MongoDB. | Used for exploratory testing and inspecting data in MongoDB. |
| RestAssured | API testing framework for RESTful services in Java. | Used to write and execute API tests for currency rate endpoints. |
| Hamcrest Matchers | Library for writing readable and flexible assertions in Java tests. | Integrated with RestAssured for validating API responses. |
| Selenium | Web automation tool for testing Web UI components (future phase). | Will be used for UI automation. |
| Testcontainers | Library for creating and managing containers for testing environments (Docker-based). | Used to manage 3 containers: app container and 2 MongoDB containers. |
| Docker | Containerization platform to run the application and MongoDB in isolated environments. | Used to run the application and MongoDB instances during tests. |
| JUnit5 | Testing framework for writing and executing Java-based tests. | Used to write and run automated test cases for both API and Web UI. |
| Maven | Build automation tool for Java projects. | Used to build the project, including tests, and manage dependencies. |
| Maven surefire report plugin | Maven plugin to render reports from .xml to web interface version (HTML format). | Used to create test reports. |

# 6. Test Data Management

The management of test data is a key part of the test automation strategy. Given that the application does not handle sensitive data, the focus will be on the following approaches for creating, storing, and handling test data:

1. **Parameterized Test Data**  
   Part of the test data will be stored in a separate file (e.g., CSV, JSON, or XML) and consumed using the @ParameterizedTest annotation in JUnit. This allows for the execution of the same test with different inputs, improving test coverage and reusability. There are no current plans to destroy or modify this data as it will not change between test runs.
2. **Runtime Data Creation**  
   Some test data, such as date-dependent data or data inserted into databases during tests, will be generated during the runtime of the tests. This ensures that the data is relevant to the execution context and reflects the conditions that may exist during the actual use of the application. After test execution, any data created during runtime will be destroyed to maintain a clean testing environment and avoid clutter in databases.
3. **Data Destruction**  
   Test data created at runtime will be cleaned up and destroyed after test execution. This will include removing any records in the databases or temporary files created for testing purposes.

# 7. Test Environment

The test environment is designed to replicate the application’s real-world execution environment as closely as possible. The environment will be set up on local machines using Docker and the Testcontainers library, allowing the application and its dependencies (MongoDB) to run in isolated containers.

**Key details of the test environment:**

1. **Test Environment Setup**
   * Testcontainers and Docker will be used to create and manage the testing environment on local machines.
   * **The environment will consist of 3 containers:**
     + Application Container: The main application will be run in a container.
     + MongoDB Containers: Two separate MongoDB containers will be used to simulate the data storage environment, one for handling API request logs and the other for storing fetched data (currency rates).
2. **No Mocks**
   * Mocks will not be used in the test cases. The tests will interact with real instances of the application and MongoDB. By avoiding system interaction simplification, we ensure that the system components integrate smoothly, reflecting actual behavior and minimizing the risk of missing future production issues.
3. **Containerization Benefits**
   * Containers provide a lightweight, consistent, and easily replicable environment for running the tests across different machines and setups.
   * This approach eliminates dependencies on dedicated test servers or complex environment setup procedures, making the testing process efficient and scalable.
4. **Local Machine Testing**
   * The test suite will be executed on local developer/tester machines, using Docker to start and manage the necessary containers during the test execution.

# 8. Test Case Design

This section details the approach for designing automated test cases to ensure comprehensive coverage of application functionality. Test cases will focus on verifying the behavior of API endpoints and, later, the Web UI, following best practices for clarity, reusability, and maintainability. The design will prioritize high-risk and high-priority scenarios to maximize the efficiency and effectiveness of the test suite.

## Test Case Objectives

* Ensure comprehensive test coverage for API endpoints and Web UI.
* Validate that all functional requirements are met.
* Confirm that the system behaves correctly under expected, edge, and negative scenarios.

## Test Design Techniques

* **Equivalence Partitioning:** To reduce the number of test cases by identifying input ranges that should behave similarly.
* **Boundary Value Analysis:** To test inputs at the edges of valid ranges.
* **State Transition Testing:** For scenarios like retrying API requests or updating data in MongoDB.
* **Error Guessing:** Based on experience, identify inputs or sequences likely to cause errors.

## Categorization of Test Cases

* **API Tests:**
  + Positive cases (valid requests, valid responses).
  + Negative cases (invalid endpoints, invalid payloads).
  + Data-related tests (validation of data storage in MongoDB).
* **Exploratory Tests:**
  + Ad hoc tests to uncover unknown bugs or unusual behaviors.
* **Web UI Tests:**
  + Functional testing of the Web UI.

## Test Case Prioritization

* **High Priority:** Critical business functionality (e.g., fetching accurate rates, MongoDB validations).
* **Medium Priority:** Logs generation, secondary error-handling scenarios.
* **Low Priority:** Cosmetic details for the future Web UI.

## Tools for Test Case Design

* Test cases will be written and maintained using automation frameworks (JUnit5 and RestAssured for API testing, Selenium for Web UI).
* Parameterized tests will be used to enhance reusability and coverage.

## Reusable Components

* Common test setup (e.g., container initialization via Testcontainers).
* Data-driven testing using the @ParameterizedTest annotation in JUnit.

# 9. Automation Test Execution Plan

The automation test execution plan maps out how and when tests will be executed to ensure the application meets quality standards during its development lifecycle.

## Execution Workflow

* + API Tests:  
    API tests will be executed with each new build. The trigger point will be when the developer creates a pull request. These tests ensure that the API endpoints function correctly and any introduced changes do not break the application.
  + UI Tests:  
    UI tests will be executed for builds that successfully pass the API tests. This two-step process ensures that the underlying API functionality is stable before validating the Web UI.

## Triggering Mechanism

* + Test execution will be manual for the time being, requiring testers or developers to initiate the test suite for API and UI tests.
  + In the future, the plan may evolve to incorporate automated triggering through Continuous Integration/Continuous Deployment (CI/CD) pipelines.

## Environment Requirements

* + Both API and UI tests will be executed in the local test environment using Docker containers managed by Testcontainers.

## Execution Frequency

* + API tests: Each new build after a pull request creation.
  + UI tests: After the build passes API tests.

## Reporting and Feedback

* + Test results will be logged and shared with the team to identify issues promptly. Failed tests will be analyzed, and any identified bugs will be reported to the developer for resolution.

# 10. Defect Management

This section outlines the process and tools used to identify, report, track, and resolve defects discovered during testing.

1. **Defect Reporting**
   * Defects will be documented in a text file stored in the project repository on GitHub.
   * Each defect entry will include:
     + **Defect ID**: A unique identifier for the defect.
     + **Title**: A brief, descriptive title summarizing the defect.
     + **Description (optional)**: Detailed information about the defect, including the context in which it was found.
     + **Steps to Reproduce**: Instructions to replicate the defect.
     + **Expected Results**: A description of the correct behavior or output.
     + **Actual Results**: A description of the observed (incorrect) behavior or output.
     + **Priority**: The order in which the defect should be fixed. Priority is based on business needs and project goals and may not always align with severity.
     + **Status**: The current state of the defect.
     + **Screenshot Link** (optional): A Google Drive link to screenshots illustrating the defect.
2. **Defect Lifecycle**
   * **Statuses**:
     + **Open**: The defect has been reported and requires developer attention.
     + **In Progress**: The defect is actively being worked on by the developer.
     + **In Test**: The developer has completed the fix, and the defect is ready for retesting by the tester.
     + **Closed**: The defect has been fixed and verified by the tester.
   * If the defect is not resolved during retesting, the status will be updated back to **In Progress** for further investigation.
3. **Defect Communication**
   * The **key tester** will notify the **key developer** about the defects that need developer’s attention via team chat.
   * Once a fix is ready, the **key developer** will notify the **key tester** to initiate verification.
4. **Defect Retesting**
   * The **key tester** will verify the defect fix by rerunning the related test case(s).
   * If the fix is successful, the defect will be marked as **Closed**. If not, it will be reopened with an updated **In Progress** status in the GitHub file.
5. **Screenshot Management**
   * If visual evidence is needed to describe a defect, screenshots will be stored on Google Drive.
   * A link to the screenshot(s) will be included in the defect entry in the GitHub text file.

# 11. Metrics and Reporting

This section outlines the reporting approach and key metrics that will be tracked to assess the quality and effectiveness of the test automation process.

## Reporting

* **HTML Reports**:  
  Test execution reports will be generated automatically using the **Maven Surefire Report Plugin**. These reports will provide:
  + A summary of the executed test cases.
  + Details of passed, failed, and skipped tests.
  + Any errors or exceptions encountered during test execution.
  + Test execution time for individual test cases and the entire suite.
* Reports will be shared with the team after each test suite execution for review and issue tracking.

## Metrics

The following metrics will be tracked to monitor and optimize the test automation process:

Here’s the revised **Metrics and Reporting** section:

This section defines the key metrics that will be used to monitor the efficiency, effectiveness, and reliability of the test automation process. These metrics provide actionable insights for optimizing test coverage, improving defect management, and ensuring the stability of the test suite.

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric** | **Explanation** | **Calculation** | **Target Value** |
| **Test Execution Time** | Measures the time taken to execute the test suite, helping identify bottlenecks and inefficiencies. | Total time taken from the start to the end of test suite execution. | Execution time within acceptable range (e.g., ≤ 10 minutes for API tests). |
| **Pass Rate** | Indicates the percentage of test cases that pass, ensuring the stability and correctness of the application. | PassRate(Pass Rate (%) = (Number of Passed Tests / Total Executed Tests) × 100 | ≥ 95% for stable builds. |
| **Defect Reopen Rate** | Tracks the percentage of defects reopened after being marked as resolved, highlighting the quality of fixes. | DefectReopenRate(Defect Reopen Rate (%) = (Number of Reopened Defects / Total Closed Defects) × 100 | ≤ 5%. |
| **Test Case Coverage** | Measures the proportion of application features covered by automated tests, ensuring adequate testing scope. | Coverage(Coverage (%) = (Number of Features Tested / Total Features) × 100 | ≥ 90%. |
| **Test Stability** | Indicates the reliability of tests by measuring the percentage of consistently passing tests. | Stability(Stability (%) = (Stable Tests / Total Tests) × 100 | ≥ 95%. |

By tracking these metrics, the team can ensure the test automation process remains efficient, reliable, and aligned with project goals.

# 12. Risk Management

This section identifies potential risks that could impact the test automation process and outlines strategies to mitigate them. Each risk is assessed based on its **probability**, **severity**, and overall **likelihood** (calculated as the product of probability and severity).

* **Probability** is rated on a scale from 1 to 10, where 1 indicates the risk is very likely to occur, and 10 indicates the risk is very unlikely.
* **Severity** is rated on a scale from 1 to 5, where 1 represents the highest impact and 5 represents the lowest impact.
* **Likelihood** is the multiplication of Probability and Severity, giving an overall risk assessment score.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Risk Name | Probability (1–10) | Severity (1–5) | Likelihood (P × S) | Symptom | Mitigation Plan | Contingency Plan |
| Environment Setup Challenges | 5 | 2 | 10 | Delays in setting up Docker containers or Testcontainers. | Allocate time for environment setup testing; create detailed setup documentation. | Seek support from online communities or the Testcontainers library team. |
| Test Data Limitations | 8 | 3 | 24 | Tests fail due to inconsistent or insufficient test data. | Predefine reusable test data files; automate runtime data creation and cleanup. | Manually review and adjust test data to resolve issues. |
| API Changes | 4 | 1 | 4 | Tests fail after unexpected changes to bank APIs. | Regularly monitor API documentation for updates; version-control API endpoints. | Update test cases promptly to align with the new API changes. |
| Flaky Tests | 5 | 3 | 15 | Tests fail intermittently due to timing issues or external dependencies. | Add appropriate waits; isolate flaky tests; improve synchronization mechanisms. | Investigate failures, disable flaky tests temporarily, and rework them. |
| Knowledge Gaps | 6 | 3 | 18 | Delays in test creation due to lack of familiarity with tools or frameworks. | Schedule time for training and experimentation; maintain comprehensive documentation. | Consult documentation, tutorials, or experts for rapid problem resolution. |
| Limited Team Resources | 6 | 2 | 12 | Overloaded team members lead to delays in deliverables. | Define priorities clearly; focus on high-value tests first. | Reallocate tasks; extend deadlines if necessary. |
| Dependency on External Factors | 7 | 4 | 28 | Network or third-party service outages disrupt testing. | Use local mock APIs for critical paths during outages (where applicable); monitor external service health. | Resume testing once the external issue is resolved; log skipped tests. |
| Test Script Maintenance | 5 | 3 | 15 | Frequent updates to tests are needed due to application changes. | Write modular and reusable scripts; use version control for test cases. | Allocate time for regular maintenance; address critical tests first. |
| No Dedicated CI/CD Integration | 3 | 4 | 12 | Delays in triggering test executions. | Create a clear process for manual test execution and scheduling. | Gradually move towards CI/CD integration as resources allow. |
| Priority Misalignment | 6 | 3 | 18 | Delays in test automation due to conflicting development priorities. | Regularly sync with the developer to align priorities; define clear goals for automation efforts. | Escalate priorities to management or stakeholders for resolution. |

\*

This format provides clear quantitative assessments of the risks and ensures a structured approach to mitigating and managing them. Let me know if any further refinements are needed!

# 13. Test Deliverables

This section outlines the key deliverables that will be produced as part of the test automation process. These deliverables ensure transparency, traceability, and effective communication among stakeholders.

1. **Test Plan for Test Automation**
   * This document (current) serves as a roadmap for the test automation process, detailing the scope, objectives, tools, and approaches for testing the application.
2. **Test Scripts**
   * Automated test scripts developed in Java for:
     + **API Testing**: Scripts to verify the functionality and reliability of the API endpoints.
     + **UI Testing**: Scripts to validate the behavior and usability of the planned Web UI.
   * These scripts will be reusable, scalable, and designed to cover key application features.
3. **Test Reports**
   * Comprehensive reports generated after each test execution, providing:
     + Summary of executed tests (passed, failed, skipped).
     + Detailed results with timestamps, logs, and any observed issues.
     + Execution metrics such as test duration and coverage.
   * Reports will be shared with the team for analysis and decision-making.
4. **Bug Reports**
   * A centralized record of identified defects maintained in a text file on GitHub.
   * Each defect entry will include a description, steps to reproduce, expected vs. actual results, priority, status, and links to supporting screenshots (if applicable).

These deliverables will collectively support the development and maintenance of a robust and effective test automation process while ensuring that quality objectives are met.

# 14. Timeline and Milestones

This section describes the key milestones and estimated timelines for the test automation process. These milestones provide a structured approach to completing the test automation deliverables.

**Milestones and Timelines**

1. **Write Test Plan and Finalize Application Requirements**
   * **Objective**: Create a comprehensive test plan (this document) and finalize the application requirements to ensure clarity and alignment between testing and development teams.
   * **Estimated Timeline**: 2 weeks.
2. **Complete API Test Suite**
   * **Objective**: Develop a robust API test suite to verify the functionality, reliability, and consistency of the application's API endpoints.
   * **Activities**:
     + Learn and implement new libraries such as Testcontainers.
     + Set up the API test framework (e.g., JUnit, RestAssured).
     + Write and refine API test scripts.
   * **Estimated Timeline**: 4 weeks.
3. **Create Web UI Automation**
   * **Objective**: Develop UI test automation scripts to validate the functionality and usability of the planned Web interface.
   * **Activities**:
     + Set up a UI test framework (e.g., Selenium).
     + Write and execute UI automation scripts.
   * **Estimated Timeline**: 2 weeks.

**Flexibility in Timelines**

The timelines for each milestone are flexible to accommodate unforeseen challenges or adjustments in priorities. The team will regularly review progress and adapt the schedule as needed to ensure the successful delivery of test automation.