***Test Plan - Vnext Integrations***  
***Created by*** [@Mohan Kamalathan - Viva Goals](mailto:v-mohanka@microsoft.com)

|  |
| --- |
| 1. *Introduction* 2. *Why Automate Tests? The ROI* 3. *Scope of Testing* *3.1 In Scope* *3.2 Out of Scope* 4. *Integration points - High Level Test Coverage* *4.1 Generic Scenarios* *4.2 Excel* *4.3 ADO* *4.4 Planner* *4.5 PowerBI* 5. *Testing Approach* *5.1 Unit Testing - ViTest* *5.2* *API Functional Testing – PostMan* *5.3* *UI Automation – PlayWright* 6. *Defect Reporting Procedure* *6.1 Defect Logging* *6.2 Defect Assignment* *6.3 Defect Triage* *6.4 Defect Resolution* *6.5 Defect verification / validation* *6.6 Defect Closure* *6.7 Defect Reporting Metrics* *6.8 Continuous Improvement* 7. *CI/CD - Test Execution* *7.1 Automated Tests on* *PRs & Commits* *7.2 UI Automated Tests Integrations with Main* *7.3 Automated Testing on Every Release cycle* *7.4 Synthetic & Nightly Tests for Stability of Integration points*  *7.5 Integration with Azure DevOps Pipelines* 8. *Roles and Responsibilities*  *8.1* *SDET Team* *8.2 Development Team* *8.3 Product / Stakeholders*  *8.4 Process /* *POD Owners – Each Integrations* 9. *Appendix* *9.1 Sources* *9.2 Additional Resources / References* |

***1. Introduction***

This document outlines a comprehensive test plan for the Integrations Part of the upcoming Vnext VNEXT application with various external applications, encompassing Excel, Ado, PowerBI and Planner.

The plan emphasizes leveraging the following Test libs and Programming Languages

* Playwright for automating UI and APIs - Typescript Binding
* JMeter for performance testing - Java Binding
* Postman for API Testing
* Unit Testing – Using ViTest

***2. Why Automate Tests? The ROIs***

Investing in test automation for Vnext integration functionalities presents a multitude of benefits that significantly enhance the development and testing process.

Here's a breakdown of the key reasons to automate tests and the return on investment (ROI) it delivers:

1. Regression Testing Efficiency: Regression testing, which involves re-testing previously validated functionalities to ensure they still work after changes, is a crucial part of software maintenance. Automating regression tests ensure that all existing functionalities continue to work as expected with each new release or update.
2. Enhanced Testing Scope: Automated testing can check more types of test cases than manual testing. They can execute repetitive tests, edge cases, and complex scenarios consistently and repeatedly. This enhanced test coverage ensures better quality assurance, reducing the likelihood of critical bugs slipping into production.
3. Scalability of Tests: As software projects grow in complexity and size, manual testing becomes increasingly challenging to scale. Automated tests, on the other hand, can easily scale to accommodate larger codebases, multiple platforms, and frequent releases.
4. Enhanced Developer Productivity: Test automation empowers developers to identify and fix defects early in the development cycle, before they escalate into larger issues. By integrating automated tests into continuous integration/continuous deployment (CI/CD) pipelines, developers receive immediate feedback on code changes, allowing them to iterate and improve code quality more efficiently.

***3. Scope of Testing***

This test plan covers both functional and non-functional aspects of the Integrations part of the Vnext Application.

***3.1 In Scope***

* E2E Automation Test Coverage – For the Vnext Integrations points Excel, ADO, Planner and PowerBI.
* Synthetic, Nightly and Regression Test Suites Creation – Using PlayWright
* API Testing Using Postman
* Unit Testing Using Vitest

***3.2 Out of Scope***

* E2E API Automation using Playwright
* Performance Testing Using JMeter

Items listed as "Out of Scope" will not be addressed during the initial phases of development but will be considered in subsequent iterations of the development life cycle.

***4. Integration points - High Level Test Coverage***

This section details the functionalities to be tested for each integrated application with the Vnext Application.

This Sections also has High level Test Scenarios for the integration points of the Vnext   
Integrations.

***4.1 Generic Scenarios:***  
  
**Creating Objectives:**

1. User should be able to create new Objectives within an organization.
2. Objectives should have a title, description, start date, end date, and owner.
3. Objectives can be categorized or tagged for better organization.
4. Each Objective should be assigned a unique identifier for reference.

**Tracking Metrics:**

1. Users should be able to define metrics associated with each Objective to track progress.
2. Metrics could include numerical values, percentages, or qualitative assessments.
3. Users should be able to update metric values regularly to reflect progress.
4. There should be visibility into historical metric data to track trends over time.

**Setting Goals:**

1. Users should be able to set specific and measurable goals aligned with each Objective.
2. Goals should be achievable within the given time limit of the Objective.
3. Goals may have milestones or checkpoints to monitor progress.
4. Users should receive notifications or alerts when goals are achieved, or milestones are reached.

**Admin Disable Integration:**

1. Admin navigates to the admin settings tab in the VNEXT application.
2. Admin finds the list of integration points and Chooses a Specific App Integration
3. Admin disables a Specific integration point for a user
4. When User creating a new goal with integration in VNEXT, Disabled integration option should not be available.
5. Test re-enabling disabled integration and verifying its availability.

**Data Syncing:**

1. Integration should automatically fetch data from external applications and synchronize it with corresponding Objectives in the VNEXT system.
2. Data syncing should be scheduled at regular intervals to ensure up-to-date information.
3. There should be mechanisms to handle errors or inconsistencies during data syncing.
4. Users should have visibility into the status of data syncing processes.

**Data Mapping and Transformation:**

1. Users should be able to define mappings between data fields from external applications and fields within the VNEXT system.
2. There should be options for data transformation or manipulation during the mapping process.
3. Users should be able to preview mapped data before finalizing the integration setup.

***4.2 Excel:***

**Setup Integration:**

1. User initiates the integration setup process within the Vnext application.
2. User selects Excel as the source application for data integration.
3. User provides authentication credentials and access permissions to the Excel file.
4. Integration module establishes a connection to the Excel file and verifies access.

**Mapping Data:**

1. User specifies the data fields within the Excel file to be mapped to Objectives within the VNEXT system.
2. The integration module displays a preview of the Excel data and allows the user to select relevant fields.
3. User maps Excel data fields to corresponding fields within the VNEXT application such as Objective title, description, and metrics.

**Syncing Data:**

1. The integration module fetches data from the specified Excel file based on the defined mapping.
2. Data from Excel is transformed, if necessary, to match the format expected by the VNEXT system.
3. Synced data is then populated into the corresponding Objectives and metrics within the VNEXT application.
4. Test handling of errors during data syncing.

***4.3 Azure Dev Ops:***

**Setup Integration:**

1. User initiates the integration setup process within the VNEXT application.
2. User selects Azure DevOps as the source application for data integration.
3. User provides authentication credentials and access permissions to the Azure DevOps project.
4. Integration module establishes a connection to the Azure DevOps project and verifies access.

**Mapping Data:**

1. User specifies the data fields within Azure DevOps work items to be mapped to Objectives within the VNEXT system.
2. The integration module displays a preview of Azure DevOps work items and allows the user to select relevant fields.
3. User maps Azure DevOps work items attributes to corresponding fields within the VNEXT application such as Objective title, description, and progress indicators.

**Syncing Data:**

1. Integration module fetches work items from the specified Azure DevOps project based on the defined mapping.
2. Work item data is transformed, if necessary, to match the format expected by the VNEXT system.
3. Synced work item data is then populated into the corresponding Objectives within the VNEXT application.
4. Test handling of errors during data syncing.

***4.4 Planner***

**Setup Integration:**

1. User initiates the integration setup process within the VNEXT application.
2. User selects Planner as the source application for data integration.
3. User provides authentication credentials and access permissions to the Planner workspace.
4. Integration module establishes a connection to the Planner workspace and verifies access.

**Mapping Data:**

1. User specifies the data fields within Planner tasks to be mapped to Objectives within the VNEXT system.
2. The integration module displays a preview of Planner tasks and allows the user to select relevant fields.
3. User maps Planner task attributes to corresponding fields within the VNEXT application such as Objective title, description, and progress indicators.

**Syncing Data:**

1. Integration module fetches tasks from the specified Planner workspace based on the defined mapping.
2. Task data is transformed, if necessary, to match the format expected by the VNEXT system.
3. Synced task data is then populated into the corresponding Objectives within the Vnext application.
4. Test handling of errors during data syncing.

***4.5 PowerBI***

**Setup Integration:**

1. User initiates the integration setup process within the VNEXT application.
2. User selects PowerBI as the source application for data integration.
3. User provides authentication credentials and access permissions to the PowerBI dashboard.
4. Integration module establishes a connection to the PowerBI dashboard and verifies access.

**Mapping Data:**

1. User specifies the data fields within the PowerBI dashboard to be mapped to Objectives within the VNEXT system.
2. The integration module displays a preview of the PowerBI data and allows the user to select relevant fields.
3. User maps PowerBI data fields to corresponding fields within the VNEXT application such as Objective metrics and key performance indicators.

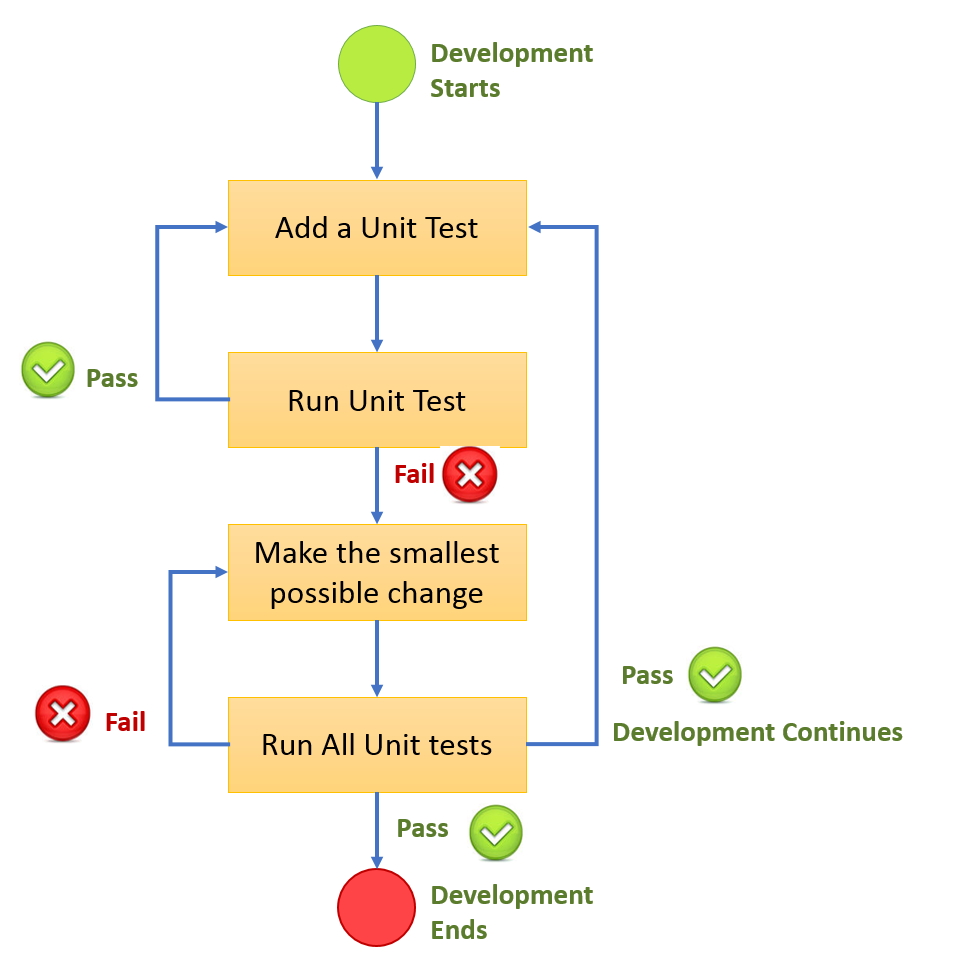
**Syncing Data:**

1. The integration module fetches data from the specified PowerBI dashboard based on the defined mapping.
2. Data from PowerBI is transformed, if necessary, to match the format expected by the VNEXT system.
3. Synced data is then populated into the corresponding Objectives and metrics within the VNEXT application.
4. Test handling of errors during data syncing.

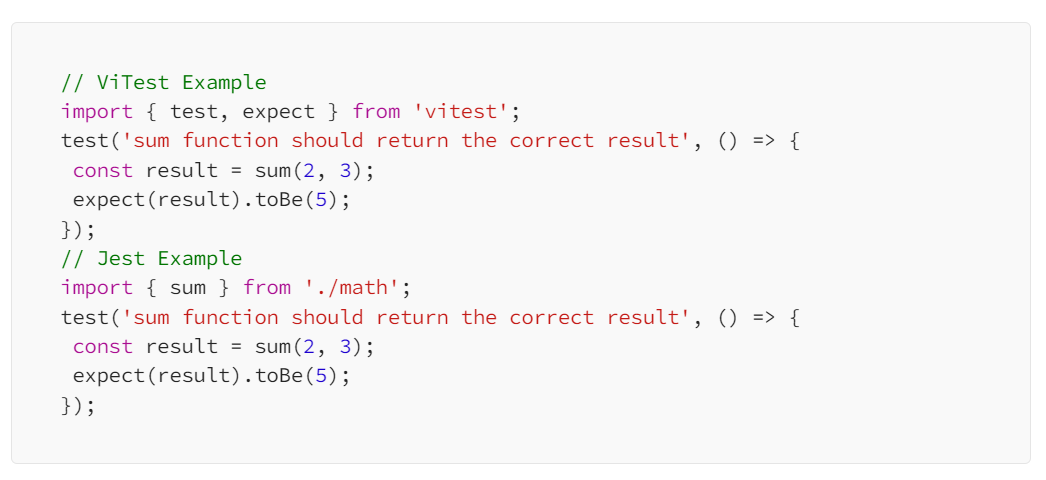
***5. Testing Approach***  
  
  
***5.1 Unit Testing – ViTest***

**Overview:**   
Unit testing is akin to scrutinizing individual building blocks to ensure they seamlessly integrate into the larger structure. Vitest serves as our tool of choice for this meticulous examination.

**Flow Of Unit Testing:**

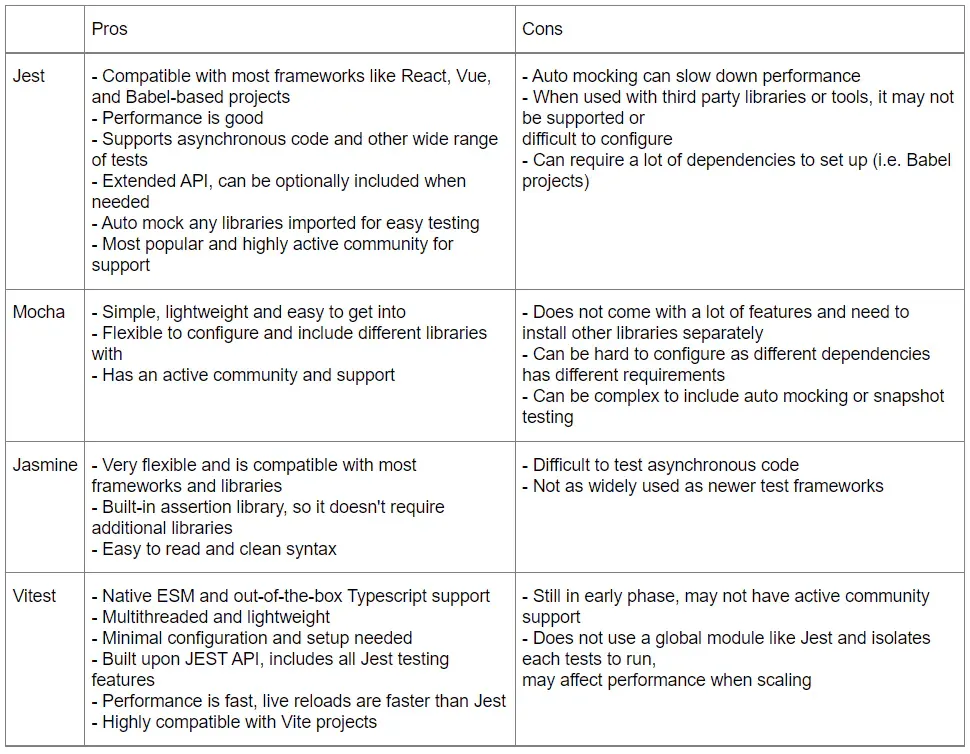


**Implementation:**   
  
Our development team meticulously craft focused unit tests using Vitest, meticulously assessing each component of our VNEXT application in isolation. This process ensures that every cog in the machine functions precisely as intended.

**Why Vitest ? The Reason**  
  
When it comes to testing your JavaScript code, the landscape offers a plethora of options. Jest and ViTest are two remarkable options among the most common ones.   
  
ViTest provides a concise and readable syntax, making it easier to write and understand test cases. With its clean and intuitive API, you can express your test expectations in a natural language-like manner.   
  
Let’s compare a basic test case written in ViTest and Jest:  
  


* ViTest works smoothly with TypeScript, using its type of system to detect mistakes and provide useful feedback during testing.

|  |
| --- |
| typescript // ViTest Example import { test, expect } from ‘vitest’;interface Person {  name: string;  age: number; }test(‘should validate the type of a person object’, () => {  const person: Person = { name: ‘John’, age: 30 };  expect(person.name).toBeType(‘string’);  expect(person.age).toBeType(‘number’); });// Jest Example import { Person } from ‘./types’;test(‘should validate the type of a person object’, () => {  const person: Person = { name: ‘John’, age: 30 };  expect(typeof person.name).toBe(‘string’);  expect(typeof person.age).toBe(‘number’); }); |

* Here is a quick comparison of the Unit Testing libs available in the Market.  
    
  

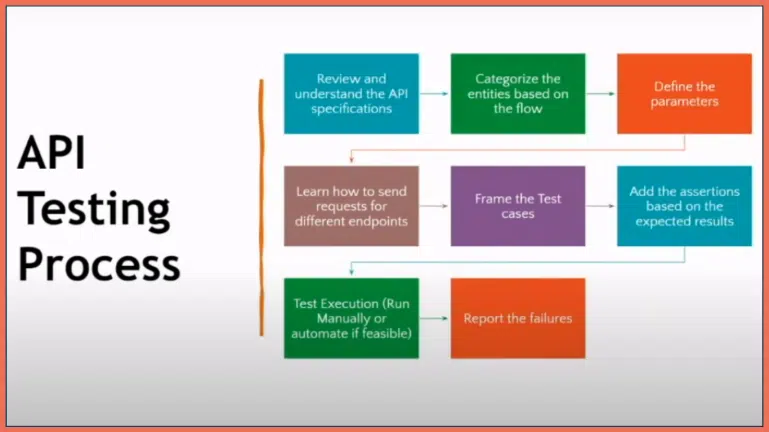
**Benefits:**  
  
***Early Bug Detection***: By identifying and addressing issues at the unit level, we mitigate the risk of larger, more complex problems arising later in the development process.  
  
***Enhanced Code Quality***: Through diligent unit testing, our codebase remains clean and maintainable, facilitating smoother collaboration and future enhancements.

***5.2 API Functional Testing – PostMan***

**Overview:**    
  
Postman is a platform that allows users to perform API functional testing, which is a method of checking the functionality and behavior of APIs by following a systematic procedure. It verifies that APIs fulfill the given criteria, handle different inputs properly, and generate expected outputs. Postman helps teams to design, run, and automate API tests, making their testing workflows more efficient and enhancing the quality of their software products.

**Flow of API Functional Testing:**  
  
The flow of API testing typically follows these key steps:

1. **Test Planning:** Define test scenarios and cases based on API requirements.
2. **Test Execution:** Send requests to API endpoints, validate responses, and handle assertions.
3. **Result Analysis:** Analyze test outcomes, identify issues, and generate reports.
4. **Iterate:** Refine tests based on findings and repeat the testing process for continuous improvement.



**Implementation:**

To test an API using Postman, you simply send an HTTP request to the API endpoint and view the response in the Postman interface. You can also use Postman’s built-in tools to verify the response data, such as checking the status code or even the response payload to validate the response data to ensure that the API is functioning as expected.

let's dive deeper into each step and how to implement API functional Testing using Postman:

**Test Environment Setup:**

1. Install and configure Postman on the testing machine.
2. Set up environments within Postman for different testing environments (PPE, DF, SIP, PROD and EU-PROD).
3. Configure environment variables such as base URLs, authentication tokens, and other necessary parameters.

**Create Test Suites in Excel:**

1. Define test scenarios and test cases in an Excel spreadsheet.
2. Each test case should include details such as API endpoint, request method, request payload, expected response, and any preconditions or dependencies.
3. Organize test cases into logical groupings based on API functionality or modules.

**Import Test Suites into Postman:**

1. Use Postman's "Collection Runner" feature to import test suites from Excel into Postman.
2. Map Excel columns to Postman variables for dynamic test execution.
3. Ensure proper formatting and structure of imported test suites to align with Postman's collection format.

**API Testing with Postman:**

1. Execute API test suites using Postman's Collection Runner.
2. Send requests to API endpoints defined in the test cases.
3. Validate responses against expected outcomes specified in the test cases.
4. Verify status codes, response headers, and response body content for correctness.
5. Handle assertions and validations within Postman to automate verification of test results.

**Data-Driven Testing:**

1. Utilize Postman's data files feature to perform data-driven testing.
2. Separate test data from test logic to improve maintainability and scalability.
3. Import data files (e.g., CSV, JSON) into Postman and iterate over each dataset during test execution.
4. Parameterize requests and assertions based on data values for comprehensive test coverage.

**API Chaining and Workflow Testing:**

1. Test complex API workflows and scenarios by chaining multiple requests together.
2. Define sequence of requests and dependencies within Postman collections.
3. Validate intermediate responses and ensure proper flow of data between API calls.
4. Implement conditional logic and scripting within Postman to handle dynamic workflows.

**Error Handling and Negative Testing:**

1. Include test cases to validate error handling mechanisms of the APIs.
2. Send requests with invalid input parameters or unauthorized access tokens to trigger error responses.
3. Verify that error responses contain appropriate status codes, error messages, and error details.
4. Test edge cases and boundary conditions to uncover potential vulnerabilities or unexpected behavior.

**Reporting and Analysis:**

1. Generate test reports within Postman to summarize test execution results.
2. Analyze test reports to identify pass/fail status, execution times, and any issues encountered during testing.
3. Export test results in various formats (e.g., JSON, HTML) for documentation and sharing with stakeholders.
4. Use insights from test reports to prioritize bug fixes, performance optimizations, or further testing iterations.

**Documentation and Knowledge Sharing:**

1. Document test cases, test suites, and testing methodologies for future reference.
2. Share best practices and lessons learned with team members to improve overall testing efficiency.
3. Provide training sessions or workshops to onboard new team members and empower them with knowledge of API functional testing using Postman.

### **Why Postman:**

Postman, which used to be a browser plugin for validation, is now the preferred tool for QA experts who work on developing, testing, and managing APIs.

* **Easy-to-Use Interface:**   
  Postman has a simple interface for creating and running API tests, making it accessible for both technical and non-technical users to operate.
* **Extensive Feature Set:**   
  Postman provides a variety of features such as environment management, data-driven testing, API chaining, and reporting, meeting different testing requirements.
* **Automation Capabilities:**   
  Postman enables the automation of API tests, allowing teams to incorporate testing into their CI/CD pipelines and improve their development workflows.
* **Collaboration and Sharing:**   
  Postman supports collaboration among team members by enabling them to share collections, work together on test scripts, and give feedback on test outcomes.
* **Community Support:**   
  Postman has a large and engaged community of users, offering access to resources, tutorials, and best practices for API testing.

**Postman Vs Rest Assured — API Testing :**  
  
Here is a quick comparison of the API Testing libs Rest Assured with the Postman:

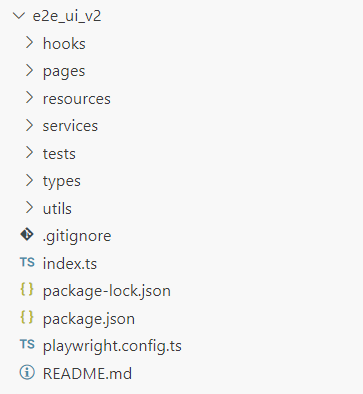
|  |  |  |
| --- | --- | --- |
| **Features** |  |  |
| Type of Tool | An application with a Userfriendly interface for API testing. | A Java DSL library used for testing RESTful APIs in code. |
| Primary Use | Manual and Automated API Testing | Automated API Testing in Java-based environments |
| language | Independent of programming languages; uses its own scripting (JavaScript) | Java |
| Ease of Learning | Easy to start with due to its GUI. | Requires knowledge of Java and understanding of library method |
| Test Creation | Tests are created using a graphical interface and scripting. | Tests are written in Java code, using BDD-style or traditional style |
| Integration with Code | Not directly integrated with code; more standalone. | Tightly integrated with the code; can be part of the codebase. |
| Environment Support | It has built-in support for environment variables and easy switching. | Environment setup must be coded. |
| Parameterization | Supports parameterization through its interface and scripting | Supports parameterization using Java code. |
| CI/CD Integration | Can integrate with CI/CD pipelines via Newman (command-line runner). | Can be integrated as part of the regular build process. |
| Reporting | Built-in reporting and supports various formats with additional tools | Limited built-in reporting but can use Java reporting tools. |
| Mock Servers, Monitors, etc. | Provides features like mock servers, monitors, etc. | Does not provide these features natively; relies on external tools |
| Data-Driven Testing | Supports data-driven testing through external data files | Supports data-driven testing using Java constructs. |
| Authentication Support | Supports various authentication mechanisms with easy setup | Supports authentication but requires manual coding |
| Use Case | Suitable for both manual testing and automation. Good for API exploration and quick tests. | Best for automation in Java projects and for integrating API tests with application code. |

***5.2 UI Automation – PlayWright***

**Overview:**

In the context of Vnext integrations, the testing approach will focus on implementing UI automation using Playwright.   
  
Leveraging the existing UI framework developed with a Test-Driven Development (TDD) approach using Playwright and TypeScript from the Vcurrent application's main branch will be instrumental in accelerating the testing process for Vnext.

**Migration of Existing Framework:**

The first step involves migrating the existing UI automation framework, which includes all necessary components, such as test scripts, utilities, and configurations, to the Vnext's main branch.   
  
This migration process sets up a solid foundation for building UI automation for Vnext.  
  
Here is an illustration of Folder Structure for Existing Automation Framework from Vcurrent:  
  


**Creation of E2E Folder:**  
  
A new "End-to-End (E2E)" folder will be created within the Vnext's main branch to house the migrated UI automation framework.   
  
This folder serves as the designated location for storing all UI automation-related artifacts and scripts for Vnext.  
  
**Establishment of Page Object Model (POM):**  
  
The migrated framework will adopt the Page Object Model (POM) design pattern same as the Vcurrent.  
  
The Existing Automation Framework Consists of Locator, Backup Locator and Page Action methods already established for Vcurrent.  
  
We are going follow the same POM approach as this is the standard approach in Test Automation.  
  
This approach encapsulates page-specific elements and page action functionalities within dedicated page objects, promoting reusability and readability of test scripts.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
For Example,

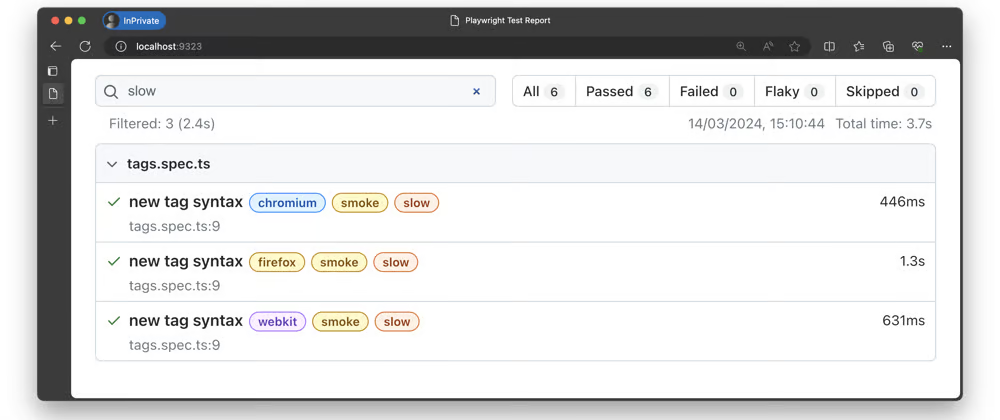
|  |  |
| --- | --- |
| **Element Locator Logic** | **Page Action Methods** |
| public elements = {  self: ():IElement => {  return {  label: "Self",  elementLocator: {  locator: "//div[contains(@class,'new-connection')]",  locatorType: LocatorType.XPATH  },  backupLocators: [  {  locator: '//div[@ng-controller="ConnectionsController"]//svg-icon[contains(@name,"integration")]',  locatorType: LocatorType.XPATH  }  ]  }  }, | async clickOnNextButton():Promise<void> {  const logger = startStep("clickOnNextButton()", this.actions.logLabel);  await this.actions.isElementVisible(this.elements.next());  await this.actions.click(this.elements.next());  await this.actions.waitForElementToDisappear(this.elements.self());  logger.end();  } |

**Test Suite Categorization:**

Tests will be categorized into various suites based on their purpose and scope:

* **Regression Tests:** Comprehensive tests aimed at ensuring existing functionalities remain unaffected by new developments.
* **Smoke Tests:** Quick, high-level tests to verify basic functionality and detect critical issues early in the development cycle.
* **Sanity Tests:** Tests targeting core functionalities to validate the application's basic functionality after each build or deployment.
* **Nightly Tests:** Scheduled tests executed overnight to cover a broader range of functionalities and scenarios.
* **Synthetic Tests:** Simulated tests designed to mimic real user interactions and scenarios to monitor application performance and reliability.

**Tagging Test Scripts:**

Each test script will be tagged according to its associated test suite (e.g., regression, smoke, sanity, nightly, synthetic).   
  
Tagging eases test organization, execution, and reporting, enabling targeted testing based on specific requirements and goals.  
  
  
  
**Testing Workflow:**   
  
**Local Testing Environment Setup:**

1. SDET sets up a local testing environment on their development machine, configuring necessary dependencies and ensuring compatibility with the Vnext application.

**Test Case Execution:**

1. SDET executes the suite of test cases locally using the existing UI automation framework powered by Playwright.
2. Test cases cover various scenarios, including regression, smoke, sanity, nightly, and synthetic tests, validating critical functionalities and use cases.

**Result Analysis and Validation:**

1. SDET analyzes test results to find failures, anomalies, or regressions, ensuring that the application meets quality standards and functional requirements.
2. Detailed logs, screenshots, and error messages aid in root cause analysis and issue resolution.

**PR Submission:**

1. Upon successful completion of local testing, SDET prepares and submits a pull request (PR) having the changes related to Vnext integrations  
   .
2. The PR includes relevant documentation, test reports, and updates to the automation framework, ensuring comprehensive review and validation.

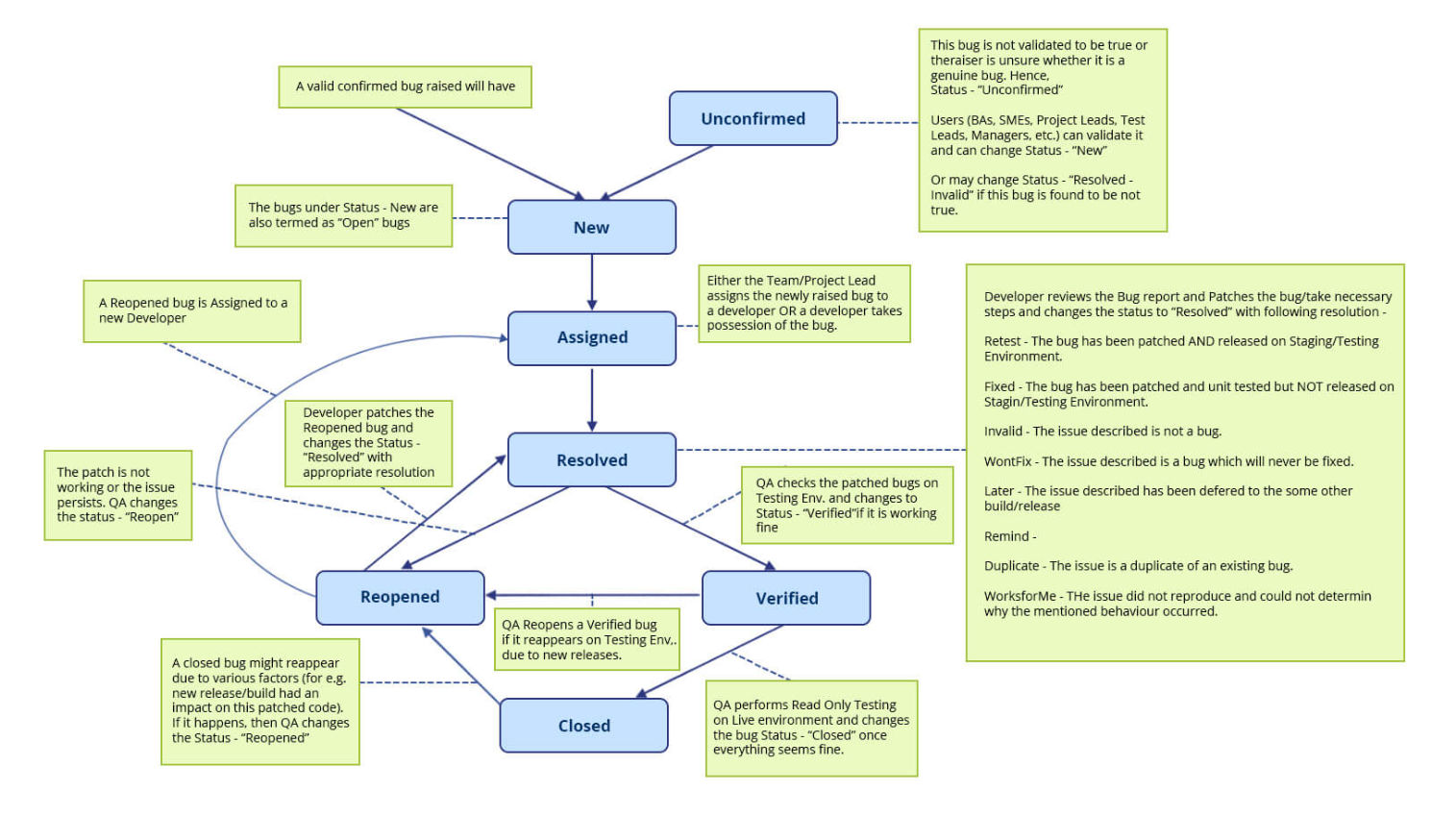
**CI/CD Integration:**

1. The PR triggers automated CI/CD pipelines configured in Azure DevOps, starting a series of automated build, test, and deployment processes.
2. Automated tests, including UI automation tests using Playwright, are seamlessly integrated into the CI/CD pipeline, validating changes and ensuring application integrity.
3. Test results are reported back to the PR and integrated into the CI/CD workflow, providing real-time feedback and visibility into the quality and stability of the application.

**Benefits of Playwright:**

Playwright is chosen as the automation tool for UI testing due to its numerous advantages:

* **Cross-Browser Support:** Playwright provides support for testing web applications across multiple browsers (Chrome, Firefox, WebKit) with consistent APIs, ensuring comprehensive browser compatibility testing.
* **Fast Execution:** Playwright's fast and reliable execution engine enables rapid execution of test cases, leading to shorter feedback cycles and faster time-to-market.
* **Powerful Features:** Playwright offers a rich set of features for UI automation, including robust selectors, device emulation, network interception, and parallel execution, empowering testers to create robust and scalable test suites.
* **First-Class TypeScript Support:** Playwright's native support for TypeScript aligns well with the existing framework's TypeScript implementation, enhancing code readability, maintainability, and type safety.

***6. Defect Reporting Procedure***  
  
In the Azure DevOps environment, the defect reporting procedure is streamlined and efficient, leveraging the features and capabilities of Azure Boards and Azure Repos.   
  
This procedure ensures that defects are logged, tracked, and resolved in a systematic manner, facilitating effective collaboration between development, testing, and stakeholders.  
  
Here is an illustration of Bug/Defect Life Cycle for better understanding:  
  


***6.1 Defect Logging***

* When a defect is identified during testing, the SDET logs the defect in Azure Boards, providing detailed information such as:   
    
  1. Description of the defect  
  2. Steps to reproduce  
  3. Severity and priority  
  4. Assigned team member  
  5. Screenshots or attachments to illustrate the issue

***6.2 Defect Assignment***

* Once logged, the defect is assigned to the appropriate team member responsible for resolution.
* With Azure Boards, we can quickly assign and monitor bugs, making sure everyone knows who is responsible and what is going on at every stage of the bug's lifecycle.

***6.3 Defect Triage***

* In a regular triage meeting, the defect is reviewed by the development and SDET teams, along with stakeholders.
* The severity and priority of the defect are reassessed based on its impact on the project timeline and user experience.

***6.4 Defect Resolution***

* The assigned team member works on resolving the defect by making necessary code changes.
* Using Azure Repos, the code changes are committed and linked to the corresponding defect for traceability.

***6.5 Defect Validation / Verificatio***

* Once the code changes are implemented, the SDET verifies the defect fixed in a test environment.
* If the fix is successful, the SDET updates the status of the defect in Azure Boards accordingly.

***6.6 Defect Closure***

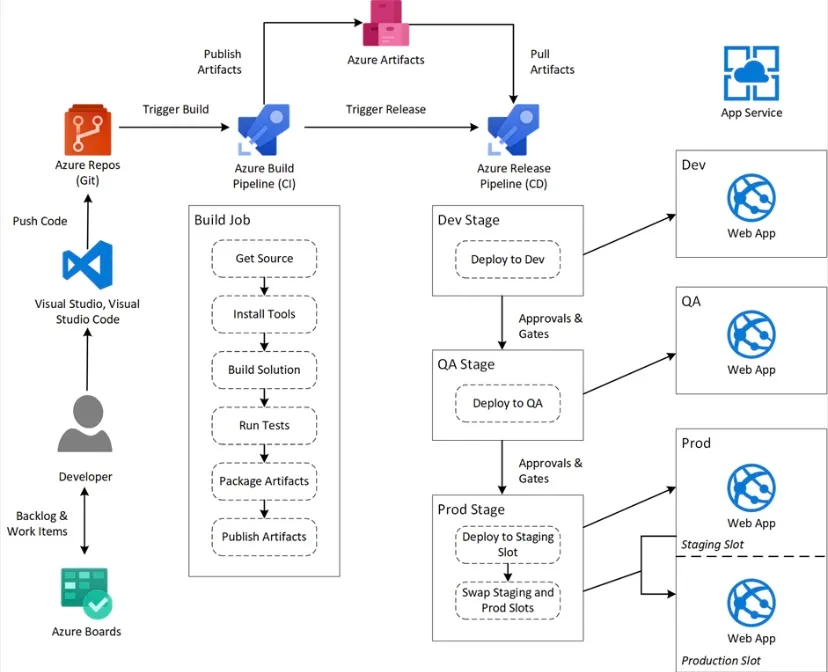
* Once verified, the defect is marked as resolved in Azure Boards.
* The resolution details, including the code changes and test results, are documented for future reference.

***6.7 Defect Reporting and Metrics***

* Azure DevOps provides built-in reporting and analytics capabilities to track defect metrics.
* Managers and stakeholders can access customizable reports and dashboards to monitor defect trends, resolution times, and overall project health.

***6.8 Continuous Improvement***

* Periodic SDLC reviews should be done to find out how to improve the defect detection and fixing process.
* The process will be improved and adjusted over time by using the feedback from retrospectives as a guide.

***7. CI /CD - Test Execution***  
  
Our CI/CD testing strategy ensures software reliability and quality across various stages of development and deployment.  
  
We use Azure DevOps pipelines to run automatic tests at important phases of the CI/CD process, so we can identify and resolve problems quickly.  
  
Here is an illustration of CI/CD architecture with Azure DevOps:  
  


***7.1 Automated Testing on Every PRs & Commits***

* After every PR submission and commit, automated testing is triggered to perform Smoke, Sanity, and Regression tests.
* These tests are essential for quickly identifying potential regressions or issues introduced by code changes, ensuring that only stable and reliable code is merged into the main branch.

***7.2 UI Automated Tests Integrations with Main***

* UI automation tests are developed locally using Playwright, ensuring robust test scripts that accurately simulate user interactions.
* After every PR submission and commit of UI automation code, Smoke, Sanity, and Regression tests are executed to validate the stability of the test automation framework.
* This approach ensures the reliability of UI automation tests and detects any framework-related issues early in the development process.

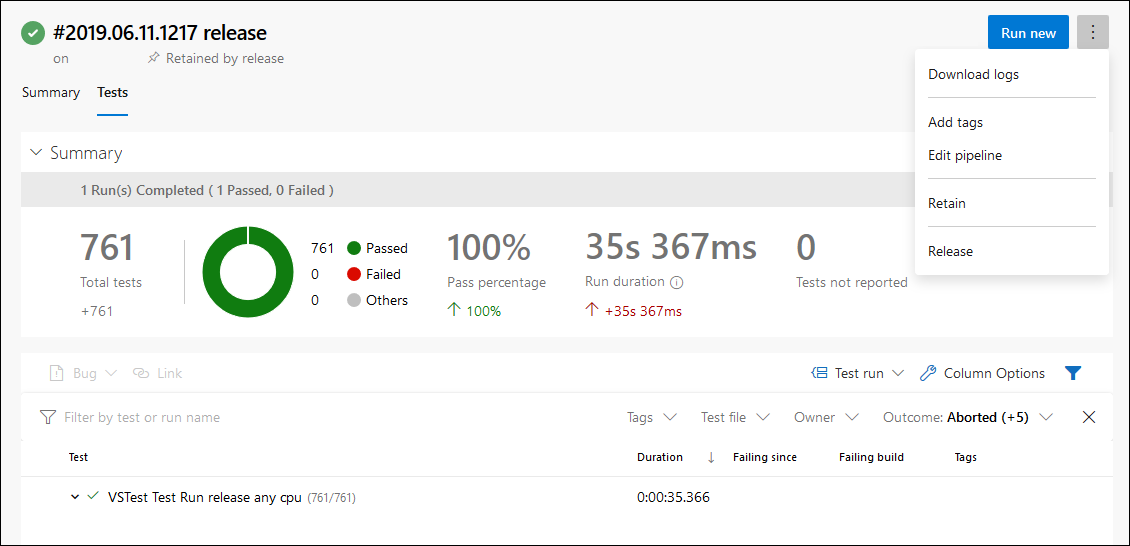
***7.3 Automated Testing on Every Release Cycle:***

* Upon every release, automated testing is conducted across multiple environments, including Pre-Production (PPE), Development Framework (DF), Staging in Production (SIP), and Production (PROD) environments.
* Smoke, Sanity, and Regression tests are executed to ensure the stability and integrity of each release before deployment to production or staging environments.
* In E2E Automation Framework We will apply Tags to distinguish Smoke, Sanity and Regression Test Cases.

***7.4 Synthetic & Nightly Tests for Stability of Integration Points***

* Synthetic and Night Tests are scheduled to run daily at specific times to verify the stability of integration points.
* Synthetic tests focus on verifying whether integration points are enabled and functioning correctly, providing early detection of any failures or issues.
* Night tests comprehensively assess the integration points' stability, ensuring that they continue to operate smoothly and reliably.
* Synthetic tests are scheduled to run at \_\_Specific Time\_\_ daily.
* Night tests are scheduled to run at \_\_Specific Time\_\_ daily.

***7.5 Integration with Azure DevOps Pipelines***   
  
Test results are automatically captured and displayed within Azure DevOps, providing visibility into the health and status of each test run in every environment.

Here is an Illustration of a Test report from Azure DevOps :  
  
  
  
Automated tests are seamlessly integrated into Azure DevOps pipelines, allowing for streamlined execution and reporting across all environment  
  
***8. Roles & Responsibilities***  
  
This section outlines the cohesive responsibilities within the test automation process for the Vnext Integrations, led by SDEs and SDETs. The SDET team collaboratively ensures the quality, functionality, and performance of the application through a systematic and comprehensive testing approach.

***8.1 SDET Team***  
  
***Primary Responsibility***  
  
The SDET team is entrusted with designing, implementing, and executing robust testing strategies to validate the Viva Goals Vnext Integrations functionality and performance.

***Specific Tasks***

* Develop end-to-end (E2E) UI tests using Playwright, adhering to the Page Object Model (POM).
* Design and execute API Functional and Integration tests using Postman.
* Establish Test-Driven Development (TDD) practices for E2E testing in collaboration with developers, guiding development with comprehensive test coverage.
* Conduct Functional Manual testing to supplement automated tests, focusing on edge cases and user experience.
* Coordinate closely with the development team to prioritize and address identified issues promptly, ensuring timely resolution.
* Provide feedback on the effectiveness of test automation processes and suggest improvements to enhance overall quality and efficiency.

***8.2 Development Team***

***Primary Responsibility***  
  
The development team, in conjunction with the SDET team, is accountable for writing clean, maintainable code and ensuring the functionality of individual components through unit and Integration testing.  
  
***Specific Tasks***

* Write unit tests using the Vitest unit testing library to verify the behavior of code modules.
* Collaborate closely with the SDET team to ensure comprehensive test coverage aligns with development goals and user requirements.
* Address any issues identified during unit and Integration testing promptly and efficiently to maintain code quality and integrity.

***8.3 Product / Stakeholders***

***Primary Responsibility***   
  
Stakeholders play a pivotal role in providing guidance, feedback, and support throughout the test automation process, ensuring alignment with business objectives and user needs.

***Specific Tasks***

* Define acceptance criteria and user requirements to guide test automation efforts, ensuring they meet business objectives and quality standards.
* Review and approve test plans, strategies, and results to validate their alignment with organizational goals and user expectations.
* Provide continuous feedback and guidance to the SDET team, facilitating informed decision-making and perfecting the testing process.

***8.4 Process / POD Owners - Integrations Points***

In our Vnext application, below are the designated Process / POD Owners responsible for overseeing and managing each integration points and processes :

|  |  |
| --- | --- |
| **Process / Integration Point** | **POD Owner** |
| Engineering & Dev Management | Ankush Jain |
| Product / Stakeholder | Aarushi Arora |
| Excel Development | Goutham Surendra Inchapuram |
| Planner Development | Ram Narendar |
| PowerBI Development | Rajesh K |
| Azure DevOps Development | Ashish Mittal |
| Unit Testing | Dev POD owner of each Integration point |
| Integration Testing | Dev POD owner of each Integration point |
| API & Functional Testing | Mohan Kamalanathan |
| UI & API automation | Mohan Kamalanathan |

***9. Appendix***

The Appendix section typically serves as a repository for supplementary materials that provide additional context, information, or resources relevant to the testing process.

***9.1 Sources:***

The Sources section of the test plan serves as a comprehensive reference for all libraries utilized throughout the testing process.

**UI Testing: PlayWright**

Documentation**:** <https://playwright.dev/docs/intro>

**Integrations Testing: Postman**

Documentation: <https://learning.postman.com/docs/introduction/overview/>

**Unit Testing: ViTest**

Documentation: <https://vitest.dev/guide/>

**Performance Testing: JMeter**

Documentation: <https://jmeter.apache.org/usermanual/index.html>

***9.2 Additional Resources / References***  
  
***9.2.1 High level Test Coverage of Vcurrent Integrations:***

**Excel**

***Data Exchange:***

* Test the import of VNEXT data from well-formatted Excel spreadsheets.
* Verify the export of VNEXT data to Excel spreadsheets for easy sharing and analysis.

***Data Mapping and Validation:***

* Test the accuracy of data mapping between VNEXT s and corresponding Excel sheet columns.
* Validate error handling mechanisms for scenarios involving invalid data formats or inconsistencies in the Excel sheet.

**Planner:**

***Task Management***

* Verify the creation and synchronization of tasks between the Vnext application and Microsoft Planner.
* Test the ability to link tasks in Planner to relevant VNEXT s for improved traceability and context.
* Ensure progress updates in Planner accurately reflect the status of associated VNEXT s.

**Power BI**

***Data Visualization:***

* Verify the transfer of VNEXT data to Power BI for insightful visualizations.
* Test the ability to create custom dashboards and reports within Power BI based on VNEXT data.

***Drill-Down and Analysis:***

* Ensure users can effectively drill down into specific VNEXT s from Power BI visualizations for deeper analysis.
* Test the accuracy and clarity of data representation within Power BI reports for informed decision-making.

**Looker**

***Advanced Data Analytics:***

* Verify the seamless transfer of VNEXT data to Looker for advanced data analysis and exploration.
* Test the ability to create custom data models and visualizations within Looker for deeper insights into VNEXT performance.

***Data Security and Governance:***

* Ensure data security measures are maintained during the integration with Looker.
* Test the effectiveness of data governance controls to ensure accurate and reliable data analysis for informed decision-making.

**SmartSheet**

***Project Collaboration and Planning:***

* Verify the integration enables collaborative planning and execution of tasks aligned with VNEXT s using SmartSheet.
* Test the ability to assign tasks to team members and track progress towards VNEXT completion within SmartSheet.

***Dependency Management and Resource Allocation:***

* Ensure the integration eases the management of task dependencies crucial for achieving VNEXT s.
* Test the ability to effectively distribute resources within SmartSheet based on VNEXT priorities.

**VSTS (Visual Studio Team Services)**

***Project Management and Tracking***

* Verify the synchronization of VNEXT s with VSTS for aligning project tasks and milestones with overall Goals.
* Test the ability to track project progress towards VNEXT s within VSTS dashboards for improved visibility and communication.

***Bug Tracking and Issue Resolution:***

* Ensure the integration eases the creation and tracking of bugs or issues that hinder VNEXT achievement within VSTS.
* Test the ability to link bugs or issues to specific VNEXT s for prioritization and resolution.

**Salesforce**

***Sales Performance Management:***

* Verify the integration of VNEXT s with Salesforce data to analyze the impact of VNEXT s on sales performance.
* Test the ability to visualize sales team progress towards VNEXT s within Salesforce dashboards.

***Lead Management and Pipeline Tracking***

* Ensure the integration eases tracking how VNEXT s influence lead generation and sales pipeline progression.
* Test the ability to identify potential roadblocks in achieving sales related VNEXT s through Salesforce data insights.

**Google Sheets**

***Data Sharing and Collaboration:***

* Test the functionality for sharing VNEXT data with users through Google Sheets.
* Verify real-time collaboration features, allowing multiple users to edit and update VNEXT data within Google Sheets.

***Data Validation and Security:***

* Ensure data integrity is maintained during collaboration in Google Sheets.
* Test the effectiveness of access control mechanisms to secure sensitive VNEXT information.

**Jira**

***Issue Tracking:***

* Verify the creation of Jira issues based on information captured within the VNEXT s.
* Test the linking of Jira issues to relevant VNEXT s for comprehensive issue tracking and progress monitoring.

***Status Updates and Workflows:***

* Ensure that status updates in Jira issues accurately reflect the progress of associated VNEXT s.
* Test the compatibility of VNEXT workflows with existing Jira workflows to streamline issue management.