Test Plan: Online Marketplace Vendor Management System Enhancement

# Overview

Project Name: Online Marketplace Vendor Management System Enhancement

Client: Online Market Place (OMP)

Date: [Insert Date]

# Project Description

The Online Marketplace Vendor Management System Enhancement aims to enhance the vendor management system for an online marketplace. It includes features such as product listing management, bulk import/export, data mapping, real-time previews, product versioning, advanced search and filtering, inventory forecasting, multi-language support, and inventory alerts.

# Test Objectives

* Ensure the functionality and reliability of the enhanced vendor management system.
* Validate that vendors can effectively manage product listings and utilize new features.
* Verify data mapping works at the catalogue level and maintains consistency.
* Confirm that inventory management, search, and forecasting functionalities work as expected.
* Ensure multi-language support is seamless and accurate.
* Validate bulk import/export and versioning capabilities.
* Confirm that API endpoints produce correct responses and handle various scenarios.
* Test the security and authentication aspects of the system.
* Verify that notifications are sent to vendors as intended.

# Scope

Here are the

### User Interfaces (UI)

* Vendor Login
* Vendor Dashboard
  + Product Search Bar
  + Product Preview Widget
  + Vender Notification Pannel
  + Inventory Tracking and Forecast Widget
* Vendor Product Import Page
* Vendor Product Export Page
* Vendor Product Management Page

### API

* **Authentication and Authorization**
  + /api/auth/login
  + /api/auth/logout
  + /api/auth/check
  + /api/auth/reset-password
  + /api/auth/refresh-token
* **Admin APIs**
  + /api/users
  + /api/vendors
* **Vendor-Specific APIs**
  + /api/vendors/{vendor\_id}/products
  + /api/vendors/{vendor\_id}/orders
  + /api/vendors/{vendor\_id}/orders/{order\_id}
  + /api/vendors/{vendor\_id) /inventory/{product\_id}
  + /api/vendors/{vendor\_id}/inventory/{product\_id}/adjust
  + /api/vendors/{vendor\_id}/catalogues/{catalogue\_id}/import
  + /api/vendors/{vendor\_id}/catalogues/{catalogue\_id}/export
  + /api/vendors/{vendor\_id}/inventory/{product\_id}/forecast
  + /api/vendors/{vendor\_id}/catalogues/{catalogue\_id}/data-mapping
  + /api/notifications/vendors

### Data Store:

* Vendor Table
* Catalogue Table
* Products Table

## Testing Approach

### Unit Testing:

**Unit Testing of UI Components**

Unit testing of UI components involves testing individual elements and interactions within the user interface, such as React components in a web application.

**Unit Testing Strategy**

1. **Component Testing**: Test each UI component independently to ensure it renders correctly and functions as expected. Use mocking and stubbing techniques to isolate components from external dependencies.
2. **Event Handling Testing**: Verify that event handlers (e.g., click, change, submit) attached to UI elements trigger the expected behavior or state changes.
3. **Snapshot Testing**: Capture and compare component snapshots to detect unexpected changes in rendering output.
4. **Prop Testing**: Validate that components correctly handle and display props, including various combinations and edge cases.

**Comprehensiveness of UI Unit Testing**

The minimum degree of comprehensiveness desired for UI unit testing includes:

* **Coverage**: Aim for high test coverage to ensure that the majority of UI components are tested.
* **Functionality**: Ensure that all critical functionality within UI components is tested.
* **Edge Cases**: Test edge cases, boundary values, and error scenarios.

**Techniques to Determine Comprehensiveness**

To determine the comprehensiveness of UI unit testing, the following techniques will be used:

1. **Code Coverage Analysis**: Use code coverage tools to measure the percentage of code covered by unit tests. Aim for high coverage, especially for critical components.
2. **Review and Inspection**: Conduct code reviews and inspections to assess the quality and coverage of unit tests.
3. **Traceability to Requirements**: Ensure that unit tests are traceable to specific requirements outlined in the SRD.

**Unit Testing of API Endpoints**

Unit testing of API endpoints involves testing individual API methods or endpoints, typically using frameworks like Jest for JavaScript-based APIs.

**Unit Testing Strategy**

1. **API Endpoint Testing**: Test each API endpoint independently, covering various request and response scenarios.
2. **Input Validation Testing**: Verify that API endpoints handle input validation correctly and reject invalid requests.
3. **Error Handling Testing**: Test error handling scenarios, ensuring that appropriate error responses are returned.

**Comprehensiveness of API Unit Testing**

The minimum degree of comprehensiveness desired for API unit testing includes:

* **Coverage**: Test all API endpoints, including various HTTP methods (GET, POST, PUT, DELETE).
* **Request Variations**: Test different request scenarios, including valid, invalid, and edge cases.
* **Error Scenarios**: Ensure that error scenarios, such as 404 Not Found or 500 Internal Server Error, are tested.

**Techniques to Determine Comprehensiveness**

To determine the comprehensiveness of API unit testing, the following techniques will be used:

1. **Code Coverage Analysis**: Measure code coverage for API endpoints to ensure that a significant portion of the code is tested.
2. **Review and Inspection**: Conduct code reviews and inspections to assess the quality and coverage of unit tests for API endpoints.
3. **Traceability to Requirements**: Ensure that API unit tests are traceable to specific requirements outlined in the SRD.

**Unit Testing of Data Stores**

Unit testing of data stores involves testing interactions with the relational and NoSQL databases to ensure data integrity and consistency.

**Unit Testing Strategy**

1. **Database Query Testing**: Test database queries and operations to ensure they retrieve, insert, update, and delete data correctly.
2. **Data Validation Testing**: Validate that data stored in the databases complies with the defined schema and constraints.

**Comprehensiveness of Data Store Unit Testing**

The minimum degree of comprehensiveness desired for data store unit testing includes:

* **Coverage**: Test critical database operations and queries used by the application.
* **Data Consistency**: Ensure that data stored in the databases is consistent with the application's requirements.

**Techniques to Determine Comprehensiveness**

To determine the comprehensiveness of data store unit testing, the following techniques will be used:

1. **Database Testing Tools**: Utilize database testing tools and libraries to create unit tests for database interactions.
2. **Review and Inspection**: Conduct code reviews and inspections to assess the quality and coverage of unit tests for data stores.
3. **Traceability to Requirements**: Ensure that data store unit tests are traceable to specific requirements outlined in the SRD.

**Completion Criteria**

* Unit tests must achieve a code coverage rate of at least 80% for UI components, API endpoints, and data store interactions.
* Unit tests must pass without any failures, indicating that the tested units are functioning correctly.

### Integration Testing:

Integration testing focuses on verifying the interactions and data flow between different components, such as UI components, API endpoints, and data stores. The goal is to identify and rectify issues related to component integration, communication, and data consistency.

**Integration Testing Strategy**

1. **Component Integration**: Test the integration of UI components with each other, ensuring that they render correctly and interact as expected.
2. **API Endpoint Integration**: Verify that API endpoints communicate correctly with the UI components and with each other, including handling various request and response scenarios.
3. **Data Store Integration**: Validate the integration between the application and the data stores (relational database and NoSQL database). Ensure data consistency and integrity when interacting with the databases.
4. **Authentication and Authorization**: Test the integration of the authentication and authorization service with API endpoints, ensuring that access controls and permissions are enforced.

**Integration Testing Scenarios**

Scenario 1: User Authentication

* **Test Objective**: Verify that the authentication service integrates correctly with UI components and API endpoints for user login.
* **Test Steps**:
  1. Attempt to log in with valid credentials.
  2. Attempt to log in with invalid credentials.
  3. Verify that authenticated users can access authorized API endpoints.
  4. Login with valid credentials, change credentials, logout and attempt login with older credentials.
  5. Attempt login with valid Email ID with ‘+001’ before @ as login ID.
  6. Attempt user registration with temporary email domain clients.

Scenario 2: API Endpoint Integration

* **Test Objective**: Validate the integration between UI components and API endpoints for product management.
* **Test Steps**:
  1. Create, read, update, and delete products through API endpoints.
  2. Verify that changes made in UI components reflect correctly in the API endpoints and vice versa.
  3. Attempt accessing other vendor resources with valid login session.
  4. Attempt sending invalid request variation from UI components.

Scenario 3: Data Store Integration

* **Test Objective**: Ensure data consistency and integrity when interacting with the relational and NoSQL databases.
* **Test Steps**:
  1. Insert and retrieve data from databases.
  2. Verify that changes made in UI components and API endpoints are correctly reflected in the databases.

### System Testing

System testing assesses the entire software application as a whole, verifying that it meets the specified requirements and functions correctly in its intended environment. It encompasses end-to-end testing of the complete system, including the UI, API interactions, and data stores.

**System Testing Strategy**

1. **UI Testing**: Validate the UI components and their interactions to ensure they deliver the intended user experience.
2. **End-to-End Scenarios**: Test complete end-to-end scenarios, including user journeys, from UI interaction to API requests and data store operations.
3. **Performance Testing**: Assess the system's performance, including load testing, response times, and resource utilization, to ensure it meets performance requirements.
4. **Security Testing**: Verify the application's security mechanisms, including authentication, authorization, and data protection, to identify and mitigate security vulnerabilities.

**System Testing Scenarios**

Scenario 1: End-to-End Product Listing

* **Test Objective**: Validate the complete product listing process, including UI interactions, API requests, and data store operations.
* **Test Steps**:
  1. Create a product in the UI.
  2. Verify that the product is correctly displayed in the UI product listing.
  3. Check that the product information is accurately stored in the data stores.
  4. Check that ordering items from store reduces the availability of items from vendor stock.
  5. Verify cancelling of an order re-stock the item back to vendor stock.
  6. Verify vendor notification on any order event.
  7. Check item forecasting shows current trend of item sales.
  8. Verify Vendor can upload multiple catalogues of items with data mapping.
  9. Verify bulk upload process can recover from errors in a csv file.
  10. Verify Line items not processed are shown on vendor notification widget.
  11. Check if a single product can have new version updated without affecting older listing.
  12. Check weather import and export can be used to import and export products from multiple catalogues.

Scenario 2: Performance Testing

* **Test Objective**: Assess the system's performance under various loads and conditions.
* **Test Steps**:
  1. Simulate multiple concurrent users interacting with the UI and making API requests.
  2. Measure response times, resource utilization, and system stability.

Scenario 3: Security Testing

* **Test Objective**: Identify and address security vulnerabilities in the application.
* **Test Steps**:
  1. Test for common security issues such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).
  2. Verify that authentication and authorization mechanisms are secure and effective.

### Performance Testing

Performance testing evaluates how well the application performs under normal and expected loads. It measures response times, throughput, and resource utilization to ensure the application meets performance requirements.

**Performance Testing Strategy**

1. **Load Testing**: Gradually increase the load on the system to assess its capacity. Monitor response times and resource consumption to identify performance bottlenecks.
2. **Stress Testing**: Apply extreme loads to the system beyond its specified capacity to determine how it behaves under stress. Evaluate how the application recovers from stress conditions.
3. **Scalability Testing**: Assess the system's ability to scale horizontally or vertically. Test its performance as resources (e.g., servers) are added or removed.
4. **Resource Utilization Testing**: Measure CPU, memory, and network utilization to ensure that the application optimally utilizes available resources.

**Performance Testing Scenarios**

Scenario 1: Load Testing

* **Test Objective**: Evaluate the system's response times and resource utilization under expected user loads.
* **Test Steps**:
  1. Simulate a moderate number of concurrent users interacting with UI components.
  2. Monitor response times for UI actions and API requests.
  3. Record CPU and memory utilization.

Scenario 2: Stress Testing

* **Test Objective**: Assess the system's behaviour under extreme loads and identify breaking points.
* **Test Steps**:
  1. Gradually increase the number of concurrent users beyond the system's capacity.
  2. Observe how the system handles increased load, including any degradation in response times.
  3. Monitor system stability and recovery.

Scenario 3: Scalability Testing

* **Test Objective**: Determine the system's ability to scale horizontally or vertically.
* **Test Steps**:
  1. Add or remove servers or resources dynamically while monitoring performance.
  2. Verify that the system scales efficiently and adapts to changing resource availability.

**Sequence of Events for Performance Testing**

1. **Test Environment Setup**: Prepare a dedicated test environment that replicates the production environment as closely as possible, including servers, databases, and network configurations.
2. **Test Script Development**: Performance engineers or test developers will write test scripts simulating user interactions with UI components and API endpoints. These scripts will generate synthetic data when needed.
3. **Load Generation**: Execute load testing scenarios, gradually increasing user loads while monitoring response times and resource usage.
4. **Stress Testing**: Apply extreme loads to the system to identify its breaking points and observe how it recovers from stress conditions.
5. **Scalability Testing**: Test the system's ability to scale horizontally or vertically by adding or removing resources dynamically.
6. **Monitoring and Analysis**: Continuously monitor system performance during testing. Collect and analyse metrics related to response times, throughput, and resource utilization.
7. **Issue Identification**: Identify and document performance bottlenecks, scalability limitations, or stability issues.
8. **Optimization**: Collaborate with development teams to address identified issues. Optimize code, configurations, or infrastructure as needed.
9. **Regression Testing**: Repeat performance testing after optimizations to ensure improvements are effective and do not introduce new issues.

### UAT Team and Responsibilities

The UAT team consists of individuals responsible for conducting UAT, including:

1. **User Acceptance Testers (Vendors)**:
   * **Responsibilities**: Vendors who will use the system for their product management and catalog operations.
2. **User Acceptance Testers (Administrators)**:
   * **Responsibilities**: Administrators responsible for managing vendors and overseeing system operations.
3. **Test Lead**:
   * **Responsibilities**: Oversee the UAT process, coordinate testing activities, and ensure that test scripts are executed correctly.
4. **Test Analyst**:
   * **Responsibilities**: Develop UAT test scripts, assist users in executing tests, and document and report defects.

**UAT Testing Process**

**Test Script Development**

1. **Test Script Creation**: The Test Analyst will create UAT test scripts based on user stories and scenarios outlined in the SRD. Test scripts will include step-by-step instructions for users to follow.

**Execution of Test Scripts**

1. **User Training**: The UAT team will receive training on using the system, including API interactions, data entry, and navigation.
2. **Test Environment Setup**: Ensure that the UAT environment is set up to replicate the production environment, including access to the Online Marketplace Vendor Management System.
3. **Execution of Test Scripts**: Users (Vendors and Administrators) will execute the prepared test scripts. These scripts will involve interactions with UI components, API endpoints, and data stores.
4. **Data Preparation**: Users may use available request/response data from API endpoints and generate synthetic data when necessary to simulate real-world scenarios.
5. **Documentation**: Users will document their test results, including any defects or discrepancies they encounter during testing.

**Defect Reporting**

1. **Defect Identification**: Users will report any defects, issues, or deviations from expected behavior they encounter during testing.
2. **Defect Documentation**: The Test Analyst will document and prioritize reported defects, including detailed information on how to reproduce issues.

**Regression Testing**

1. **Defect Resolution**: Development teams will address and resolve reported defects.
2. **Regression Testing**: After defect resolution, the UAT team will retest affected areas to verify that the issues have been successfully resolved and that fixes have not introduced new problems.

**Completion Criteria**

1. **Acceptance Criteria**: The UAT team will validate that the Online Marketplace Vendor Management System meets the acceptance criteria defined in the SRD.
2. **Error Frequency**: The UAT team will monitor and evaluate the frequency and severity of defects. The goal is to ensure that the system operates with a low error frequency.

**User Approval**

1. **User Approval**: Upon successful completion of UAT, the User Acceptance Testers (Vendors and Administrators) will provide formal approval or acceptance of the system.

**Sequence of Events for UAT**

1. **Test Script Preparation**: The Test Analyst creates test scripts based on user stories and requirements.
2. **User Training**: Users (Vendors and Administrators) receive training on system usage and UAT procedures.
3. **Environment Setup**: The UAT environment is configured to replicate the production environment.
4. **Execution**: Users execute the test scripts, interacting with UI components, API endpoints, and data stores.
5. **Defect Reporting**: Users report defects and discrepancies encountered during testing.
6. **Defect Resolution**: Development teams address and resolve reported defects.
7. **Regression Testing**: The UAT team retests affected areas to verify defect resolution and assess regression.
8. **Acceptance Criteria**: The UAT team validates that the system meets acceptance criteria and operates with minimal errors.
9. **User Approval**: User Acceptance Testers provide formal approval of the Online Marketplace Vendor Management System.