# ThingsBoard ****Automation Testing PoC – Documentation****

This Proof of Concept (PoC) outlines the design and implementation of an automation testing framework developed for validating both UI and API workflows of the ThingsBoard Demo Platform. The primary objective of this initiative is to establish a lightweight, scalable, and modular testing foundation that supports secure, data-driven validation using modern Python-based tools. The framework demonstrates how integrated automation can streamline end-to-end quality checks across device telemetry and dashboard interfaces in IoT platforms.

The framework is built using the following components:

* Playwright for browser-based UI automation
* Requests for testing RESTful API endpoints
* Pytest for test execution, management, and reporting

Test data, including credentials, is securely managed in an external Excel file (pwd.xlsx) and loaded dynamically using Pandas in combination with OpenPyXL, supporting a flexible, data-driven testing strategy.

### Libraries & Tools Used

The automation framework was built using the following open-source Python libraries:

* Pytest was used as the primary framework to structure, execute, and report test results.
* Playwright enabled automated browser-based UI testing across multiple browsers.
* Requests facilitated testing of REST APIs by handling authentication and endpoint validation.
* Pandas allowed easy reading and handling of structured test data stored in Excel files.
* OpenPyXL worked as the backend engine to support Excel file operations within the Pandas workflow.

|  |
| --- |
| Project Structure  ProjectThingsboard/  ├── conftest.py # Pytest fixtures (auth, headers, credentials)  ├── utils.py # Excel reading utility  ├── utils/  │ └── pwd.xlsx # Credentials Excel file (username, password)  └── tests/  ├── test\_dashboard\_ui.py # UI tests using Playwright  └── test\_api\_telemetry.py # API tests using requests |

Folder Structure Summary

The project is organized to separate test logic, utilities, and data for clarity and maintainability:

* conftest.py: Contains shared Pytest fixtures for authentication, headers, and credential management.
* utils.py: A helper script that reads login credentials from Excel and supplies them to the tests.
* utils/pwd.xlsx: Stores external test data such as usernames and passwords in a structured format.
* tests/test\_dashboard\_ui.py: Implements Playwright-based UI tests, validating user interface elements.
* tests/test\_api\_telemetry.py: Contains API test cases using the Requests library to validate telemetry endpoints

### **How These Components Work Together**

In this PoC, **Playwright** is combined with **Pytest** to automate browser-based UI workflows such as login and widget validation. For backend validation, **Requests** is used alongside **Pytest** to test API flows including authentication and telemetry retrieval. Test credentials are stored externally in an Excel file and read using **Pandas** with **OpenPyXL**, allowing secure and flexible data handling. Common tasks like loading tokens and setting headers are centralized using fixtures in conftest.py, which helps reduce code duplication and improves test maintainability across both UI and API layers.

### **Test Data Flow**

Test credentials are maintained in an external Excel file located at utils/pwd.xlsx. The utils.py script uses **Pandas** to read this file and extract the necessary login details. These credentials are then passed into **Pytest fixtures** defined in conftest.py, which handle authentication, generate JWT tokens, and supply authorization headers to the API test cases. This setup ensures secure, reusable, and centralized handling of test data across the entire automation framework.

### **Test Execution Flows**

Two types of test cases are implemented in this PoC to validate both the UI and API layers of the ThingsBoard platform:

* **UI Test** (test\_dashboard\_ui.py):  
  Automates login using Playwright and verifies the presence and visibility of key dashboard widgets like Devices and Alarms. Credentials are passed dynamically via fixtures.
* **API Test** (test\_api\_telemetry.py):  
  Sends a GET request to fetch telemetry keys for a specific device using the Requests library. The test validates the HTTP response code (200 OK) and ensures expected keys (e.g., temperature) are present in the returned JSON.

### **Sample Test Logic**

The automation framework follows a behavior-driven approach for both API authentication and data validation.

* **Authentication Flow:**  
  If valid credentials are retrieved from the Excel file, a login request is made via the API. Upon successful authentication, a JWT token is returned.
* **Telemetry Validation Flow:**  
  Using the authenticated JWT token, the test queries the telemetry endpoint for a specific device and verifies that the response includes expected telemetry keys such as **temperature** or **pressure**.

### **Risks & Limitations**

While the PoC is effective for validating key UI and API flows, a few risks and limitations should be noted:

* **Credential Exposure:** Storing login details in pwd.xlsx can pose a security risk if the file isn’t properly secured or encrypted.
* **API Rate Limiting:** Frequent or parallel test executions may trigger rate limits, affecting test stability.
* **Test Data Drift:** Device IDs or telemetry keys used in the tests may become outdated, leading to failures if not regularly maintained.
* **Frontend Sensitivity:** Changes in the DOM structure can break UI tests, requiring regular updates to selectors.
* **Hardcoded Values:** Using static device IDs or endpoints reduces flexibility; parameterization is recommended for production readiness.

### **Automation Strategy**

The framework is designed with maintainability and scalability in mind. A **data-driven approach** is used, with credentials and input data managed externally in Excel for easy updates. **Reusable Pytest fixtures** handle login, token generation, and headers, promoting consistency across tests. The structure follows a clear **separation of concerns**, with UI and API test logic kept in separate modules. **Security best practices** are followed by avoiding hardcoded credentials or sensitive data in the codebase. Automated **HTML test reports** are generated for each run, enhancing visibility. The framework is **scalable and easy to extend**, and common logic is abstracted into utility functions to reduce duplication and support long-term **maintainability.**

### **Recommendations for Production Readiness**

To prepare this PoC for production use, the following improvements are recommended:

* **Secure Storage:** Move pwd.xlsx to a secrets manager or use encrypted storage to protect sensitive credentials.
* **Parameterization:** Replace hardcoded device IDs and keys with configurable environment variables or .env files.
* **Test Data Maintenance:** Regularly update and validate test data to prevent failures due to outdated inputs.
* **CI/CD Integration:** Integrate the framework with CI/CD tools like Jenkins or GitHub Actions for automated execution during pull requests or scheduled runs.
* **Platform Monitoring:** Continuously monitor ThingsBoard for UI/API changes and update selectors or endpoints as needed to keep tests reliable.

### **Test Case: test\_get\_telemetry\_keys**

**Purpose:**  
To confirm that the API endpoint correctly returns a list of available telemetry keys for a specified device, such as temperature or pressure.

**Test Flow:**

* Constructs the API request URL using a predefined device\_id.
* Uses headers from a Pytest fixture containing a valid JWT token.
* Verifies the response returns HTTP 200 OK and a valid JSON array.
* Asserts that specific keys (e.g., temperature, pressure) are included in the response.

**Expected Outcome:**  
The API should expose metadata reflecting the telemetry structure of the device, confirming active telemetry reporting.

### **Test Case: test\_fetch\_latest\_telemetry**

**Purpose:**  
To validate that the API returns the latest values for each telemetry key associated with a device.

**Test Flow:**

* Sends a GET request to the telemetry endpoint formatted with the target device\_id.
* Authenticates using headers from a Pytest fixture.
* Checks for a 200 OK response and verifies that the returned JSON object includes the expected telemetry keys.
* Ensures each key has a non-empty data set (i.e., contains one or more readings).

**Expected Outcome:**  
Confirms that the platform is delivering accurate, real-time telemetry data that can be consumed by dashboards or analytics tools.

### **Reusability & Security**

Both UI and API test cases leverage reusable fixtures defined in conftest.py, which centralize the handling of credentials and JWT token generation. This approach ensures:

* **Secure Token Handling:** Authentication tokens are generated dynamically and passed securely to test cases.
* **No Hardcoded Credentials:** Sensitive information such as usernames and passwords are kept external and never embedded in the source code.
* **Modular Test Design:** Common authentication logic is abstracted, keeping individual tests clean, maintainable, and easy to scale