ThingsBoard
Test Automation Strategy Document

Contents

1. Vision & Mission 3
2. Tool Selection Rationale 3
3. Automation Pyramid Design 3
   3.1 Unit & Integration Tests (50%) 4
   3.2 API & Contract Tests (35%) 4
   3.3 UI End-to-End Tests (15%) 4
4. Test Data Strategy 5
   4.1. Tenant Isolation 5
   4.2. Device Profiles (Parameterized JSON) 5
   4.3. Feature Flags (Progressive Rollout Controls) 5
   4.4. Simulation Modes 5
5. CI/CD Integration Approach 6
6. Test Case Traceability & Versioning 6
   6.1. Mapping Test Cases to Requirements (Reporting tool ↔ TC ID) 6
   6.2. Version-Controlled Test Suites 7
   6.3. Key QA Metrics We Track 7
7. Real-Time Data Validation 7
8. Roadmap & Maintenance 7
   8.1 Phases Approach 8

1. Vision & Mission

Vision

To establish a scalable, reliable, and maintainable automation strategy for ThingsBoard application especially those integrating ThingsBoard ensuring consistent quality across UI, APIs, and real-time telemetry pipelines in industrial automation use cases.

Mission

To deliver confidence in every ThingsBoard software release by implementing an end-to-end automated validation framework. This includes rigorous testing of device telemetry ingestion (e.g., from sensors and PLCs), industrial dashboards, and backend API flows, fully integrated into CI/CD pipelines to support agile product delivery and digital operations.

Strategic Objectives:

- Ensure system reliability for real-time industrial data flows (e.g., temperature, pressure, alarms)
- Achieve 90% automated coverage across critical UI and API layers
- Reduce manual validation cycles
- Detect telemetry issues or UI regressions

2. Tool Selection Rationale

A robust test strategy depends on picking the right tools that balance developer productivity, execution speed, and scalability. Below is a carefully curated stack tailored for ThingsBoard.

Purpose	Tool/Framework	Rationale
UI Automation	Playwright	Supports modern web apps with headless mode, smart waits, fast execution
API Testing	Python Requests	Clean, readable syntax; perfect for REST-heavy IoT platforms
Contract Validation	Postman	Excellent for API mock validation, schema enforcement, and collection re-use
Test Runner	Pytest	Pytest fixtures and modularity enhance maintainability
Reporting	HTML report	Delivers detailed HTML reports with history, trends, and screenshots
Performance	JMeter	Simulate load on telemetry endpoints; fits well in CI for nightly stress
CI/CD	GitHub Actions / Azure	easy YAML configs,
Monitoring	Azure Dashboards / Slack	Real-time test alerts and team notification; easy Allure + Jira integration

3. Automation Pyramid Design

The **Automation Pyramid** is a fundamental design principle that helps us optimize test coverage, speed, and reliability by structuring tests across different layers of the software system. For THINGSBOARD's ThingsBoard-based use case—where industrial IoT data is streamed, visualized,

and acted upon—this pyramid ensures that we test the right things at the right levels, keeping quality high and costs low.

Layer	Focus Areas	Coverage
Unit & Integration	Parsers, Rule Engines, Data Models	50%
API & Contract	REST/gRPC endpoints, telemetry push, alarms	35%
UI End-to-End	Login, Device Dashboard, Alarms Widget, Feature Toggles	15%
Real-time Validation	Dashboard widget sync, Web Socket/message polling	N/A

3.1 Unit & Integration Tests (50%)

What this means:
At the foundation of the pyramid lie unit and integration tests. These cover the smallest testable parts of the system such as telemetry parsers, edge rule chains, device payload processors, or data model transformations.
Note: Fast and reliable—these tests typically run in milliseconds. They help catch logic errors early before they ripple upward. Integration tests add confidence that internal modules (like rule engines and device profiles) interact correctly and in context**,** Validating that a temperature parser correctly converts incoming raw data or that a transformer rule engine executes business logic accurately ensures that higher layers (UI/API) behave predictably.

3.2 API & Contract Tests (35%)

What this means:
Here, we focus on validating ThingsBoard's exposed APIs and services—especially RESTful endpoints that interact with device telemetry, alarm generation, and tenant provisioning.
Note: Verifies that services are communicating as expected (e.g., GET /telemetry/values) , Ensures reliability even when the UI layer changes (API contracts are more stable) and Critical for automated integration points like DevOps pipelines or mobile apps. For example, after pushing simulated pressure values from a remote device, an API test checks that these values are stored correctly and retrievable in real time via telemetry endpoints. It helps ensure that remote diagnostics or alerts are data-backed.

3.3 UI End-to-End Tests (15%)

What this means:
These are full-flow validations from a user's perspective—simulating real user interactions with the ThingsBoard web console.
Note: Ensures critical features like login, dashboards, and alarm widgets function as expected, Catches layout and usability issues and Can validate if dynamic values (like device counts) appear correctly after ingestion. A UI test might simulate an operator logging in, viewing a transformer dashboard, and checking for active critical alarms. Though slower than unit/API tests, these provide assurance for production-readiness.
Design Decision: We intentionally keep UI automation light and focused, avoiding brittle or redundant tests. Only the most business-critical workflows (like alarms and device management) are automated at this level.
Design Goal Summary: The pyramid emphasizes "fast feedback" through automation at the code level and API level. These tests are quicker to execute, easier to maintain, and give immediate value in CI/CD pipelines. The UI and real-time validations provide confidence in user experience and operational fidelity but are scoped narrowly to avoid flakiness.

By designing our test strategy this way, we strike a balance between depth (unit), coverage (API), and real-world impact (UI + Real-time) ensuring ThingsBoard application functions are robust, responsive, and production-ready.

4. Test Data Strategy

In any automation framework—especially one involving real-time industrial data—the quality of test data defines the reliability of the tests themselves. At THINGSBOARD, where telemetry data from field devices like relays, transformers, or valve controllers is critical, the strategy for test data needs to be dynamic, isolated, and representative of real-world use cases.

Test data strategy into key pillars:

4.1. Tenant Isolation

- Each automation run provisions a fresh tenant via ThingsBoard' s APIs. Once tests are done, the tenant and its associated entities (devices, dashboards, alarms) are deleted.
- No cross-test pollution: Prevents flaky tests caused by leftover data from previous runs.
- Parallel-safe: Enables concurrent execution in CI/CD pipelines across multiple environments.
- Secure: Test tenants use mock credentials and non-production endpoints to prevent leakage.

4.2. Device Profiles (Parameterized JSON)

- We define devices using JSON templates, including metadata (e.g., "location": "Site-A"), sensor capabilities (temperature, pressure), and telemetry frequency.
- Reusable and scalable: Same profile templates can be reused across different test cases.
- Domain-specific modeling: Supports THINGSBOARD's specific device behaviours like alarm thresholds, telemetry frequency, and device states.

4.3. Feature Flags (Progressive Rollout Controls)

- Automation scripts are equipped to toggle features dynamically—such as enabling/disabling a new alarm rule or a beta widget—through pipeline variables or API configurations.
- Safe deployments: Allows testing of new features without impacting live tenants.
- Canary testing enabled: Test cases can simulate both stable and experimental environments.

4.4. Simulation Modes

We define a phased strategy to simulate realistic telemetry conditions:

Phase 1: Script-Based Simulation (MQTT/HTTP + Curl)

- Quick to implement.
- Mimics device push using: CMD
    - -curl -X POST https://demo.ThingsBoard.io/api/v1/<TOKEN>/telemetry \-H "Content-Type:application/json" \ -d '{"temperature": 72, "pressure": 14}'

- Used in CI pipelines, API testing, and basic dashboard validations.

Scenario: A smart valve in a water grid system can be connected to ThingsBoard. When water pressure increases beyond a set point, the dashboard and alarms are validated in real-time, just like in a live deployment.

5. CI/CD Integration Approach

Branching Strategy: GitFlow (feature -> develop -> release/x.y -> main)

Pipeline Stages:

Pre-check: config tests

- **Unit Tests**: Fast test cycles for business logic (<1 min)
- **API Tests**: Run test_telemetry_api.py using Pytest
- **UI Tests**: Run Playwright dashboard widget tests
- **Smoke/Load**: Execute JMeter scripts with multi-tenant load
- **Full Regression**: Scheduled nightly

Environment Matrix: Dev ➔ QA (Feature Flags) ➔ Staging

Reporting:

- reports archived per job
- integration for failures
- screenshot capture for UI errors

6. Test Case Traceability & Versioning

In large-scale industrial automation platforms like THINGSBOARD's digital systems, it's not just about **writing test cases**—it's about **knowing what you tested, why you tested it, and when it changed**. This section outlines how we bring full traceability and discipline into our test automation lifecycle.

6.1. Mapping Test Cases to Requirements (Reporting tool ↔ TC ID)

Every user story or requirement in Jira is **mapped to a unique Test Case ID**.

For example:

- TC_API_001 → Authentication API token validation
- TC_UI_003 → Dashboard alarm widget rendering

Real-world scenario:

If an API that retrieves temperature data fails in staging, the linked story and its test case ID (TC_API_003) help us track what broke, when it was last tested, and whether the failure is due to a regression or environment issue.

6.2. Version-Controlled Test Suites

What it means:

Every automation suite is stored in Git and tagged with the application release number (e.g., v2.1.0, v3.0-beta). This ensures the test logic matches the version of the software it's validating.

Why this matters:

- Enables branch-specific validation (e.g., release candidates, hotfixes).
- Makes rollbacks safer—if a feature is reverted, so is its test.
- Supports parallel development streams (legacy vs. new architecture, beta vs. stable).

6.3. Key QA Metrics We Track

A test framework should offer **insight**, not just execution. We track the following KPIs:

Metric	Purpose
Test Coverage	% of user stories or endpoints having automated tests
Test Duration	Ensures pipeline feedback stays fast (goal: <15 mins for API/UI)
Test Flakiness Rate	Detects stability issues (goal: <1% false failures)
MTTD (Mean Time to Detect)	How quickly a defect is caught post-merge
MTTR (Mean Time to Repair)	How long it takes to fix a failed test case or defect

7. Real-Time Data Validation

Manual Push Validation: curl -v -X POST https://demo.ThingsBoard.io/api/v1/<DEVICE_TOKEN>/telemetry \ --header Content-Type:application/json \ --data '{"temperature": 275, "pressure": 46}'

Validation Points:

- Confirm keys available via /api/plugins/telemetry/.../keys/timeseries
- Match UI counts with API response from /values/timeseries

Widget Testing: Playwright script asserts widget visibility & count correctness post-login.

8. Roadmap & Maintenance

We've structured our roadmap into practical, agile-driven phases that ensure rapid value delivery without compromising quality. This approach provides room for iteration, stakeholder feedback, and measurable progress.

Phase	Focus	ETA
Phase 1	Setup core framework, API tests, UI smoke, real-time ingestion	Week 1–2
Phase 2	Expand device profiles, add performance suite, Slack integration	Week 3–4
Phase 3	Full regression, test data isolation, production observability	Week 5 onward

8.1 Phases Approach

- Phase 1 ensures we validate the pipeline, core login flows, and critical telemetry ingestion right away.
- Phase 2 enriches coverage while introducing performance testing and alerting for early feedback.
- Phase 3 hardens our framework with production-grade test cycles and robust monitoring.