Project Documentation - Test & Test Automation

II.3525

Hippolyte Martin

Wandrille Bergeron

Marc-Antoine Tissot

Cyriaque de Thomassin de Montbel

1. Testing Strategy

1.1 Objectives

The main goal of our testing strategy is to ensure the reliability and stability of the application while keeping feedback loops fast.

More specifically, our tests aim to:

- Validate critical e-commerce workflows such as cart management and checkout.
- Protect business logic and API behavior using fast and deterministic tests.
- Verify user interface flows where API-level testing alone is insufficient.
- Provide measurable quality indicators, including code coverage, execution time, and pass rate.

This layered approach allows us to detect regressions early while still maintaining confidence in end-to-end user behavior.

1.2 Types of Tests Implemented

To cover the application comprehensively, we implemented several complementary types of tests:

- Unit tests focus on isolated core logic such as models, forms, and utility functions. These tests are fast and deterministic.
- API tests validate REST endpoints using pytest-django, ensuring correct responses, permissions, and error handling.
- Integration tests cover multi-step workflows where several components interact together (for example, the cart-to-payment flow).
- UI and end-to-end tests use Selenium to simulate real user behavior in the browser and validate critical user journeys.

Each test type serves a specific purpose and together they provide strong confidence in the system’s behavior.

1.3 Test Selection with Pytest Markers

To make test execution flexible and efficient, we use pytest markers defined in pytest.ini.

Available markers include:

- unit
- api
- ui
- slow
- smoke

These markers allow developers and CI pipelines to select exactly which tests to run. Examples of common selections include:

- Fast local tests (API + Unit):
  pytest -m "api or unit"
- Exclude slow or UI tests:
  pytest -m "not ui and not slow"
- Smoke tests only:
  pytest -m smoke

This approach enables quick feedback during development and more exhaustive testing in CI.

1.4 Coverage and Quality Metrics

Test execution produces several reports that help assess application quality:

- Coverage reports generated with pytest-cov:
  - Terminal output for quick feedback
  - HTML reports for detailed inspection
  - JSON reports for automated analysis
- Aggregated metrics computed using metrics_tracker.py, including:
  - Code coverage percentage
  - Test pass rate
  - Total execution time

These metrics allow us to track quality trends over time and identify areas needing improvement.

1.5 Execution Profiles

To support different workflows, we defined three standard execution profiles:

Local fast feedback loop

Runs only unit and API tests with coverage in the terminal:

```
pytest -q -m "api or unit" --cov=core --cov-report=term-missing
```

Full backend test suite (excluding UI)

Generates a full HTML coverage report:

```
pytest -m "not ui" --cov=core --cov-report=html
```

End-to-End UI tests

Runs Selenium tests (requires Chrome):

```
pytest -m ui -x
```

2. Test Automation Architecture

2.1 Overview

The project uses a hybrid Python and JavaScript testing stack:

- Backend: Django + Django REST Framework, located in core/, configured via home/settings/*.
- Backend tests: pytest with pytest-django, structured under the tests/ directory.
- UI tests: Selenium WebDriver using headless Chrome with automatic driver management.
- Frontend: React application tested using Jest (npm test).

This architecture ensures consistent testing across backend, frontend, and UI layers.

2.2 Key Components

Pytest Configuration (pytest.ini)

- Defines the Django settings module (home.settings.dev)
- Registers all test markers
- Enables coverage reporting on the core/ module

Global Fixtures (conftest.py)

Reusable fixtures simplify test setup and ensure consistency:

- api_client: DRF client for REST API calls
- user, authenticated_client: authentication helpers
- Test data fixtures (items, addresses, coupons)
- chrome_driver: headless Chrome browser
- live_server_url: URL of the Django test server

Metrics Tracking (metrics_tracker.py)

- Executes pytest with JSON coverage
- Parses results to compute coverage, pass rate, execution time, and defects
- Produces a summarized quality report

2.3 Test Organization

Tests are organized by responsibility:

- API tests: tests/test_api_*.py
  Validate REST endpoints such as authentication, user data, and catalog access.
- Unit tests: tests/test_unit.py
  Exercise core business logic in isolation.
- UI tests: tests/test_ui_selenium.py
  Drive a real browser against the Django live server.

This structure keeps tests readable and easy to maintain.

2.4 Execution Environment and Isolation

- Database: SQLite is used for testing, with migrations applied automatically.
- Isolation: Pytest fixtures and database rollbacks ensure clean state between tests.
- Browser execution: Selenium runs in headless mode to ensure CI compatibility.

2.5 Reporting and Artifacts

Test execution produces the following artifacts:

- HTML coverage report (htmlcov/)
- JSON coverage data (coverage.json)
- Optional JUnit XML reports for CI dashboards

These artifacts are collected in CI to track trends and regressions.

2.6 Extensibility

The testing framework is designed to scale:

- New domain fixtures can be added easily (orders, payments, etc.)
- Tests can be parameterized for broader coverage
- Parallel execution can be enabled using pytest-xdist in CI

3. CI/CD Pipeline Configuration

3.1 Objectives

The CI/CD pipeline aims to:

- Automatically build and test both backend and frontend
- Enforce quality gates before merging
- Produce reproducible artifacts and reports
- Enable safe deployment with rollback capabilities

3.2 Tooling

- CI platform: GitHub Actions (recommended)
- Backend: Python 3.11+
- Frontend: Node.js LTS (18+)
- Dependency caching is enabled for faster builds

3.3 Workflow Stages

The pipeline is triggered on pushes and pull requests.

Backend Test Job

- Install dependencies
- Run pytest (excluding UI and slow tests)
- Generate coverage and JUnit reports
- Upload reports as CI artifacts

Frontend Test Job

- Install Node dependencies (npm ci)
- Run Jest tests in CI mode
- Upload coverage and test reports

3.4 Environments and Secrets

- Secrets (API keys, credentials) are stored securely in the CI provider
- Environment-specific Django settings are used (dev for tests, prod for deployment)

3.5 Rollback and Observability

- Previous artifacts and images are retained for quick rollback
- Health checks run post-deployment
- Coverage trends and test stability can be monitored using tools like Codecov or SonarQube

4. Quality Metrics and Test Results

4.1 Metrics Tracked

The following indicators are used to assess quality:

- Code coverage (%) on the core/ module
- Test pass rate (%)
- Execution time (seconds)

- Defects found, classified by severity

4.2 Generating Metrics Locally

To generate backend metrics without UI tests:

pytest -m "not ui and not slow" --cov=core --cov-report=json -q

To produce a summarized report:

python metrics_tracker.py

4.3 Recommended Reports

- HTML coverage report: htmlcov/index.html
- JSON coverage data: coverage.json
- JUnit XML: reports/backend-junit.xml

4.4 Interpreting Results

- Coverage target: ≥ 80% on critical backend paths
- Pass rate: 100% required for pull requests
- Execution time: < 3 minutes for API and unit tests in CI
- Defects: analyzed by severity and root cause to prevent regressions