{EPITECH}

SEPT 25TH, 2025

QA TESTING STRATEGY

GeneWeb Migration OCaml → Python

DATE

Sept 25th, 2025

EPITECH TEAM

Version 1.0

{EPITECH}

TABLE OF CONTENTS

Executive Summary 3

Test scopes and objectives 4

Testing approach 5

Test activities and deliverables 14

Test organisation and resourcing 15

Communications approach 16

Test environment, infrastructure and tools

Lexicon

Referenced Documents

{EPITECH}

EXECUTIVE SUMMARY

GeneWeb is a free genealogy software with a web interface written in OCaml that can be used offline or as a web service. This project involves migrating the existing OCaml codebase to Python 3.11+, transforming a mature genealogical application into a modern Python-based system while preserving its core functionality and enhancing its accessibility to a broader developer community.

This document outlines the high-level test strategy for the project. The test strategy provides the framework for estimating the duration and cost of the total test effort and the scope and objectives on which these estimates are based.
Note that the test strategy is a planning tool not a living document; it will provide the starting point for test planning.
Following completion and sign-off of the Test Strategy, any changes to, risks, issues, resource needs, etc, should be managed through the standard planning and tracking processes.

The purpose of this document is to outline the high- level test strategy for the Legacy project, defining the preliminary test scope, high-level test activities, and organization, together with test management for the project. The test strategy provides the framework for estimating the duration and cost
of the testing effort at the required confidence level for the business case.

This test strategy is a planning tool that will provide the starting point for detailed test planning during the Execute stage.

{EPITECH}

TEST SCOPES & OBJECTIVES

The GeneWeb project is migrating its codebase from OCaml to Python while maintaining genealogical computation features, web interface functionalities, and data integrity. The testing will validate feature parity, performance, accessibility, and security in the new Python implementation.

In-scopes:

- **Genealogical Data Management**
  - Complete family tree data structures and relationships
  - Advanced relationship and consanguinity calculations
  - Person, family and event data models
  - Anniversary tracking and statistical tracking
- **Data Import/Export Capabilities**
  - GEDCOM format support with data preservation in notes
  - Native GW format handling
  - Database backup and restoration functionality
  - Bulk data operations & migrations
- **Web Interface and User Management**
  - Complete web interface for offline and online usage
  - Multi-tier authentication system (friends and wizards)
  - Configurable privacy controls and individual page filtering
  - Template customization system with specialized programming language

- **Search and Analytics**
  - Advanced genealogical search capabilities
  - Title of nobility search and timeline display
  - Statistical reporting and visualization
  - Relationship path finding and analysis

- **System Architecture**
  - Migration from OCaml to Python 3.11+ codebase
  - Modern web framework implementation (Flask)
  - RESTful API design for external integrations
  - Containerized deployment architecture

- D**atabase and Storage**
  - Database schema migration and optimization
  - File storage system for genealogical documents
  - Backup and recovery mechanisms
  - Data archival and retention policies

- **Security and Compliance**
  - Authentication and authorization systems
  - Data encryption and privacy protection
  - Audit logging and compliance reporting
  - GDPR and privacy regulation compliance

Out of scopes:

  - Future feature development unrelated to migration
  - Legacy OCaml code (except where used for comparison/benchmarking)
  - External plugins/modules not officially supported

Testing objectives:

- Ensure functional equivalence between OCaml and Python implementations
- Guarantee critical functionality (genealogical computation, family tree rendering, data import/export)
- Validate performance benchmarks comparable to OCaml version
- Demonstrate code passes all automated tests
- Validate automated deployment process
- Ensure compliance with security and accessibility requirements

---

{EPITECH}

TESTING APPROACH

Unit testing - Pytest 8.4+:

Pytest provides a set of useful tools for testing flask applications and projects, which is more flexible and powerful. Pytest uses a function-based syntax, which makes test writing cleaner and easier to understand.

- Powerful Features: With pytest's extensive plugin ecosystem, you can use advanced features like fixtures, markers, and parameterized testing.
- Improved Assertions: Pytest gives better error messages for failed assertions, making debugging quicker.
- Scalability: Pytest is highly scalable, allowing you to write a few simple tests for small apps or thousands of tests for large projects.

Framework Component	Technology	Version	Purpose
Core Testing Framework	pytest	8.4+	Primary test execution engine
FastAPI Integration	pytest-fastapi or httpx + pytest-asyncio	1.3+	FastAPI specific testing utilities
Coverage Analysis	pytest-cov	7+	Code coverage measurement
Parallel Integration	pytest-xdist	3.8+	Distributed test execution

Test Organization Structure:

The test organization follows genealogical domain boundaries with clear separation between unit, integration, and functional test categories.

Mocking Strategy:

So unit tests in pytest are not supposed to access the DB by default. To be able to access the DB, we use this mark from the pytest_flask package.

Mock Type	Technology	Use Case	Example
Database Mocking	pytest	Isolate database operations	Use in-memory SQLite fixture for tests
External Service Mocking	unittest.mock	GEDCOM file operations	Mock file system access
Time-based Mocking	freezegun	Privacy date calculations	Mock current date
HTTP Mocking	responses	External API calls	Mock genealogy service APIs

```
@pytest.mark.asyncio
async def test_get_person_details(async_client):
    response = await async_client.get("/api/person/123")
    assert response.status_code == 200
```

{EPITECH}

Code Coverage Requirements:

This is a code coverage report generated by the pytest-cov plugin and coverage.py library that details the percentage of code executed by the test suite using the Python testing framework Pytest. Here's why you should use Pytest to generate a code coverage report: It has a simple command line syntax for executing code coverage. It differentiates between tested and untested lines of code with a codebase.

Component Category	Coverage Target	Measurement Method	Enforcement
Core Genealogical Logic	95%	Statement coverage	CI/CD gate
Privacy Rules Engine	98%	Branch coverage	CI/CD gate
Data Models	90%	Statement coverage	Warning threshold
API Endpoints	85%	Statement coverage	Warning threshold
Utility Functions	95%	Statement coverage	CI/CD gate

Test Naming Conventions:

Test type	Naming Pattern	Example
Unit tests	test_<function>_<scenario>_<expected>	test_calculate_relationship_parent_child_returns_correct_type
Privacy Tests	test_privacy_<rule>_<condition>_<result>	test_privacy_birth_date_under_100_years_hides_data
Calculation Tests	test_calc_<type>_<input>_<output>	test_calc_consanguinity_siblings_returns_025
Integration Tests	test_integration_<workflow>_<scenario>	test_integration_gedcom_import_valid_file_success

{EPITECH}

Privacy Test Management:

```python
@pytest.fixture
def sample_person():
    """Create a sample person for testing"""
    return Person.objects.create(
        first_name="John",
        surname="Doe",
        birth_date=date(1950, 1, 1),
        birth_place="New York, NY",
        is_private=False
    )

@pytest.fixture
def private_person():
    """Create a private person for privacy testing"""
    recent_date = date.today() - timedelta(days=365 * 50)  # 50 years ago
    return Person.objects.create(
        first_name="Jane",
        surname="Private",
        birth_date=recent_date,
        birth_place="Private Location",
        is_private=True
    )

@pytest.fixture
def sample_family(sample_person):
    """Create a sample family for relationship testing"""
    mother = Person.objects.create(
        first_name="Mary",
        surname="Doe",
        birth_date=date(1925, 1, 1),
        is_private=False
    )

    return Family.objects.create(
        father=sample_person,
        mother=mother,
        marriage_date=date(1948, 6, 15)
    )

@pytest.fixture
def genealogy_tree():
    """Create a multi-generation family tree for complex testing"""
    # Grandparents
    grandfather = Person.objects.create(
        first_name="William",
        surname="Doe",
        birth_date=date(1900, 1, 1)
    )

    grandmother = Person.objects.create(
        first_name="Elizabeth",
        surname="Smith",
        birth_date=date(1905, 1, 1)
    )

    # Parents
    father = Person.objects.create(
        first_name="John",
        surname="Doe",
        birth_date=date(1930, 1, 1)
```

Integration Testing:

Integration testing focuses on verifying the interaction between genealogical components, ensuring data flows correctly through the system while maintaining privacy controls and calculation accuracy.

Integration Type	Components Tested	Test Focus	Technology
Database Integration	Models + Database	Data persistence and retrieval	pytest
API Integration	Views + Serializers + Database	HTTP endpoints and responses	pytest + httpx + pytest-asyncio
Service Integration	Business Logic + Data Layer	Genealogical workflows	pytest fixtures
External Integration	File Parsers + Validators	GEDCOM/GW format processing	Mock file systems
Integration Type	Components Tested	Test Focus	Technology

{EPITECH}

End-to-end Testing:

End-to-end testing validates complete genealogical workflows from user interaction through data processing and storage, ensuring the system works as intended for real-world genealogical researches.

Category	Test Cases	User Roles	Expected Outcomes
User Authentication	Login, role verification, session management	Public, Friend, Admin	Proper access control enforcement
Genealogical Research	Person search, relationship calculation, family tree navigation	All roles	Accurate genealogical data retrieval
Data Management	Person creation, family linking, event recording	Admin	Complete data lifecycle management
Privacy Enforcement	Birth date filtering, individual page restrictions	All roles	Privacy rules properly applied
Scenario Category	Test Cases	User Roles	Expected Outcomes

{EPITECH}

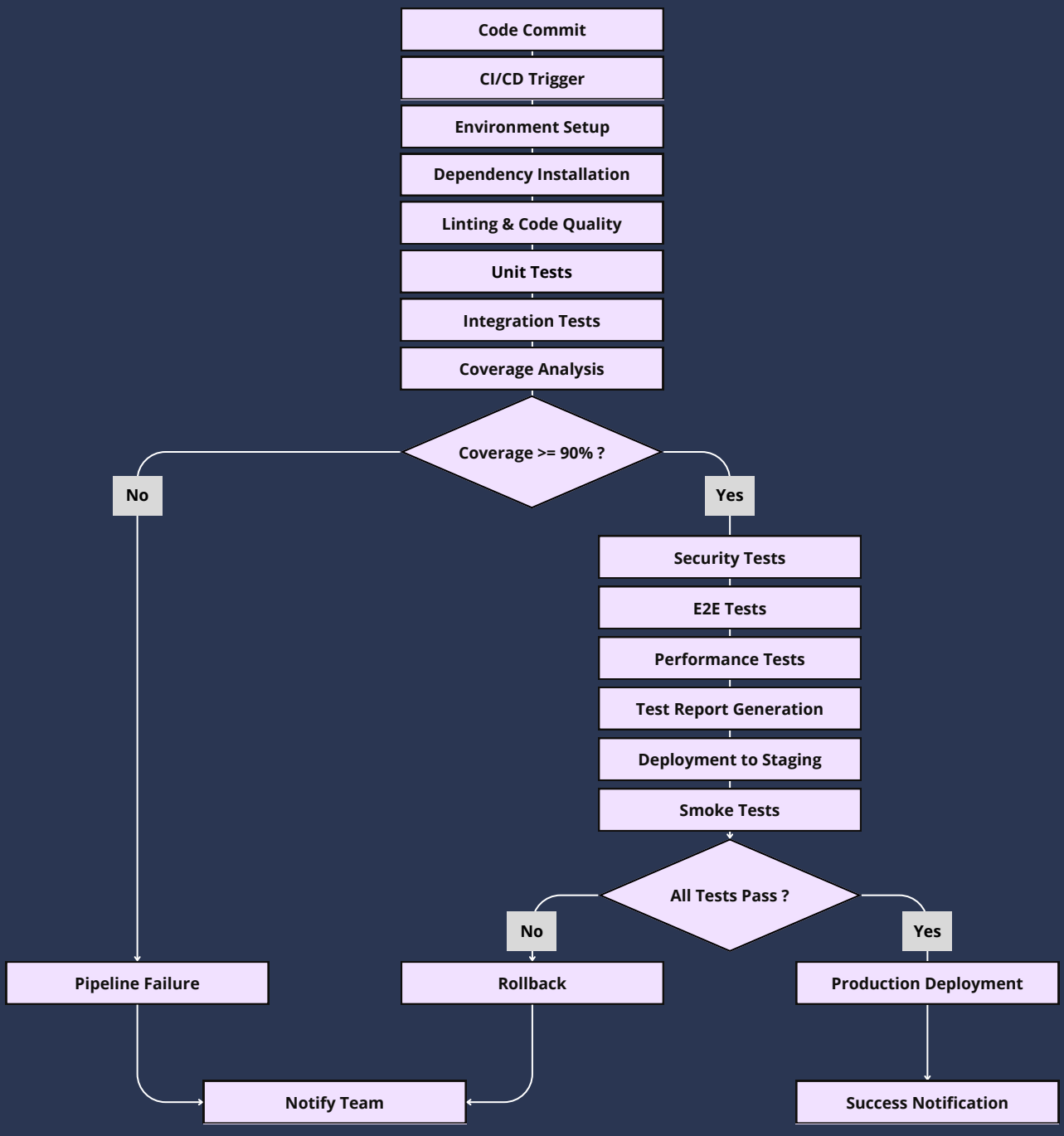
Ui Automation Approach:

Selenium, combined with Python, offers a powerful and easy-to-learn toolset for automating browser interactions. Python's simple syntax makes it ideal for quickly writing clear and maintainable test scripts. To begin, you'll need to install the Selenium WebDriver, set up a compatible browser, and learn the basics of locating web elements, interacting with them, and running test cases

```python
import pytest
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

@pytest.fixture(scope="session")
def browser_options():
    """Configure browser options for testing"""
    options = Options()
    options.add_argument("--headless")  # Run in headless mode for CI/CD
    options.add_argument("--no-sandbox")
    options.add_argument("--disable-dev-shm-usage")
    options.add_argument("--window-size=1920,1080")
    return options

@pytest.fixture(scope="function")
def browser(browser_options):
    """Provide browser instance for E2E testing"""
    driver = webdriver.Chrome(options=browser_options)
    driver.implicitly_wait(10)
    yield driver
    driver.quit()

@pytest.fixture
def authenticated_browser(browser, live_server):
    """Provide authenticated browser session"""
    # Navigate to login page
    browser.get(f"{live_server.url}/login/")

    # Perform login
    username_field = browser.find_element(By.NAME, "username")
    password_field = browser.find_element(By.NAME, "password")

    username_field.send_keys("testuser")
    password_field.send_keys("testpass123")

    login_button = browser.find_element(By.XPATH, "//button[@type='submit']")
    login_button.click()

    # Wait for successful login
    WebDriverWait(browser, 10).until(
        EC.presence_of_element_located((By.CLASS_NAME, "user-dashboard"))
    )

    return browser
```

Test Automation:

The CI/CD test automation framework integrates seamlessly with continuous integration and deployment pipelines, ensuring comprehensive testing at every stage of the development lifecycle.



{EPITECH}

Automated Test Triggers

Trigger Event	Test Suite	Execution Time	Failure Action
Code Commit	Unit + Integration	5-10 minutes	Block merge
Pull Request	Full test suite	15-20 minutes	Require fixes
Nightly Build	E2E + Performance	30-45 minutes	Alert team
Release Branch	Complete validation	60+ minutes	Block release

Parallel Test Execution:

Rich Plugin Ecosystem: Pytest boasts a vibrant ecosystem of plugins, such as pytest-xdist for parallel test execution. Pytest can run tests in parallel using plugins like pytest-xdist, significantly speeding up test execution times for large test suites.

Trigger Event	Test Suite	Execution Time	Failure Action
Code Commit	Unit + Integration	5-10 minutes	Block merge
Pull Request	Full test suite	15-20 minutes	Require fixes
Nightly Build	E2E + Performance	30-45 minutes	Alert team
Release Branch	Complete validation	60+ minutes	Block release

{EPITECH}

TEST ACTIVITITES & DELIVERABLES

Planned test activities that are to be carried out by the project to support the chosen testing approach:

Solution testing:
Activities: test environment setup, automated test execution, reporting

High Level Test Activities	Deliverables	Responsibilities
Preparation	Test Management Plan (Test Plan + Protocols)	QA Lead
Preparation	Test Case Plan (unit, integration, system)	QA Lead + Test Dev
Recording results	Pytest reports CI/CD artifacts Linter report Performance Benchmark Code snippets	QA Lead + Test Dev

User Acceptance Testing:

High Level Test Activities	Deliverables	Responsibilities
Recording results	Priority-Based UAT Scenario	QA Lead + Test Dev

{EPITECH}

TEST ORGANISATION AND RESOURCING

Structure & people requirements:

- QA Lead: oversees testing process and documentation
- Developers: Migrate code OCaml to Python, fix bugs
- QA Engineers: validate system and accessibility

Business unit	Role	Internal / external supplier	Number of resources required	Duration of assignment
Project Management	Project Manager	Internal	1 FTE	8 weeks (entire project lifecycle)
QA / Testing	QA Lead	External QA contractor	1 FTE	6 weeks (from planning → UAT completion)
Development	Python Developer	Internal	2 FTE	8 weeks (migration + testing support)
Development	DevOps Engineer	Internal	1 FTE	3 weeks (CI/CD setup + deployment automation)
User Community	UAT Testers (Public, Friend, Admin roles)	External volunteers / community contributors	3–5 part-time	2 weeks (User Acceptance Testing phase)
Technology	System Architect	Internal	1 FTE	2 weeks (architecture validation & code reviews)
Finance	Budget Controller	Internal	0.2 FTE	1 week (budget follow-up & closing)
Risk / Security	Security Analyst	External consultant	0.5 FTE	2 weeks (security & compliance audit)

{EPITECH}

COMMUNICATIONS APPROACH

- GitHub issues and pull requests for defect tracking
- Weekly project meetings

The communication approach ensures that all stakeholders, QA engineers, developers, and contributors remain aligned throughout the GeneWeb migration project. It defines the information flow, reporting cadence, and collaboration tools used to manage QA activities effectively.

Communication Objectives

- Ensure visibility and traceability of testing activities and results.
- Facilitate collaboration between development, QA, and user acceptance teams.
- Provide timely updates on test progress, risks, and issues.
- Enable data-driven decisions based on testing metrics and reports.
- Maintain transparency between internal teams and external contributors.

{EPITECH}

COMMUNICATIONS APPROACH

Communication Channels & Tools

- GitHub Issues and Pull Requests

Used for tracking defects, documenting test results, and managing code reviews. This ensures all QA-related findings are visible, version-controlled, and linked to the corresponding development work.

- GitHub Actions / Allure Reports

Automatically generate and publish test execution and code coverage reports after each CI/CD pipeline run. These reports provide real-time visibility into test outcomes and quality metrics.

- Weekly QA / Project Meetings

Scheduled once per week to review testing progress, discuss risks or blockers, and plan next steps. Attendees include the QA Lead, developers, DevOps engineer, and project manager.

- User Acceptance Testing (UAT) Workshops

Conducted during the UAT phase to walk users through acceptance scenarios, collect feedback, and verify that the migrated system meets business expectations.

- Email Summaries and QA Reports

Distributed weekly to provide stakeholders with concise summaries of QA progress, key metrics, open defects, and upcoming milestones.

- SonarQube Dashboards and CI/CD Monitoring

Provide continuous visibility into code quality, test coverage, and security findings. These dashboards serve as a live reference for both the QA and development teams.

- Retrospective Meetings

Held at the end of major testing phases to review outcomes, capture lessons learned, and define improvements for future testing cycles.

TEST ENVIRONMENT, INFRASTRUCTURE AND TOOLS

Facilities and infrastructure

- GitHub repository and Actions for CI/CD
- Developer machines for local testing

Tools

- pytest (unit/integration testing)
- Selenium (UI and accessibility testing)
- SonarQube (static code analysis, security checks)
- Allure or GitHub Actions reports (test reporting)

{EPITECH}

LEXICON

Term / Acronym	Definition
GeneWeb	Open-source genealogy software being migrated from OCaml to Python.
OCaml	The original programming language used for the legacy GeneWeb system.
Python 3.11+	Target programming language and runtime environment for the migrated system.
QA	activities to ensure software meets specified requirements.
UAT	User Acceptance Testing — final phase where end-users validate system behavior.
CI/CD	Continuous Integration / Continuous Deployment — automated build, test, and deployment pipeline.
Pytest	Primary testing framework for Python used for unit and integration tests.
GEDCOM	Standard file format for genealogical data exchange.
GW format	Native GeneWeb data format.
CI/CD Gate	Automated rule in a pipeline that prevents merging or deployment if test coverage or quality thresholds aren’t met.
Coverage	The percentage of code executed by automated tests.

{EPITECH}

LEXICON

Term / Acronym	Definition
Branch Coverage	Metric measuring how many logical branches (e.g., if/else) are executed by tests.
Statement Coverage	Metric measuring how many lines of code are executed by tests.
Fixture	Reusable test setup component in Pytest providing test data or configurations.
Mocking	Technique for simulating components (e.g., databases, APIs) during testing.
E2E Testing	End-to-End Testing — tests complete user workflows from start to finish.
CI Artifact	Output file (e.g., report, coverage result) generated by an automated CI/CD process.
SonarQube	Tool used for static code analysis, code quality, and security scanning.
Docker	Containerization platform used to run test environments consistently.
GDPR	General Data Protection Regulation — European data privacy compliance requirement.
Test Case	A single, executable test that verifies a specific feature or function.
Test Suite	Collection of related test cases executed together.
CI/CD Pipeline	Automated sequence from code commit through testing to deployment.

{EPITECH}

REFERENCED DOCUMENTS

All provided in GitHub repository docs/

FastAPI documentation: https://fastapi.tiangolo.com/
pytest-asyncio docs: https://pytest-asyncio.readthedocs.io/
httpx library: https://www.python-httpx.org/