Test Strategy

Author: Dhawanit Bhatnagar

Date: 01-August-2025

Version: 1.0

Table of Contents

1. Introduction

This document defines the **test strategy** for validating the Document Management System (DMS), ensuring it meets **functional, non-functional, security, and performance requirements** before deployment.

---

2. Objectives

- Ensure DMS is **stable, secure, scalable, and bug-free**.
- Validate **role-based access control (RBAC)** and ingestion workflows.
- Ensure **document upload, preview, and retrieval** work across all supported formats (PDF, DOC, DOCX, TXT).
- Detect **defects early** in development to reduce costs and rework.
- Validate **integration with external storage (S3/local)** and ingestion simulation.

---

3. Scope

3.1 In-Scope

- Backend APIs (Authentication, User Management, Documents, Ingestion).
- Frontend functionality (document handling, previewer, user and ingestion management).
- Integration between frontend, backend, database, and file storage.
- Supported environments: Web browsers (Chrome, Firefox, Edge).

3.2 Out-of-Scope

- Mobile application testing (will be part of future enhancement).
- AI-powered Q&A ingestion (planned in next phase).
- Load testing beyond 100k documents (future scaling).

---

4. Test Approach

1. **Agile Testing:** Testing will be iterative alongside development sprints.
2. **Shift-Left Approach:** Unit and integration tests during development phase to detect early defects.
3. **Automated Testing:** Critical API endpoints and regression tests automated using tools like Jest, Postman, and Cypress.
4. **Manual Testing:** UI, usability, and exploratory testing conducted manually.
5. **Continuous Testing:** Testing integrated in CI/CD pipeline using GitHub Actions or Jenkins.

---

5. Test Types

5.1 Unit Testing

- Scope: Functions, services, and controllers in NestJS backend.
- Tool: **Jest**
- Example: Verify /auth/login returns JWT for valid credentials.

5.2 Integration Testing

- Scope: Interaction between backend services, database, and file storage.
- Example: Upload document → Save metadata in DB → Store file locally/S3.

5.3 Functional Testing

- Scope: Verify system behavior against business requirements.
- Example: Admin can change user roles; ingestion cannot be triggered by viewers.

5.4 API Testing

- Scope: All REST APIs.
- Tool: **Postman/Newman**, **Supertest**.
- Example: Test response codes, payload validation, error handling.

5.5 UI/UX Testing

- Scope: Navigation, forms, accessibility, and usability.
- Tool: **Cypress**, **Playwright** for E2E tests.

5.6 Performance Testing

- Scope: Stress test document uploads, ingestion triggers, and retrieval.
- Tool: **k6** or **JMeter**.
- KPI: System handles 100 concurrent uploads without timeout.

5.7 Security Testing

- Scope: JWT authentication, role-based access, SQL injection prevention, file upload validation.
- Tool: **OWASP ZAP** for vulnerability scanning.

5.8 Compatibility Testing

- Scope: Cross-browser (Chrome, Firefox, Edge), screen resolutions, responsive layout testing.

5.9 Regression Testing

- Scope: Ensure new feature additions do not break existing functionality.
- Automated test suite run after every build.

5.10 UAT (User Acceptance Testing)

- Scope: Final testing by end-users or stakeholders.
- Goal: Validate real-world usability and readiness for production release.

---

6. Test Environment

- **Backend:** Node.js v20+, NestJS, PostgreSQL 15, Docker environment.
- **Frontend:** React + Vite, tested on latest browsers.
- **File Storage:** Local & AWS S3 (dev vs prod).
- Separate **QA environment** with test database and uploads directory.

---

7. Test Data Management

- Sample test users:
  - Admin, Editor, Viewer roles.
- Test documents:
  - Different formats (PDF, DOC, DOCX, TXT), various sizes (1KB–100MB).
- Anonymized data to avoid sensitive information leakage.

---

8. Test Tools

Tool	Purpose
Jest	Unit & Integration Testing
Postman/Newman	API Testing
Cypress/Playwright	E2E UI Testing
JMeter/k6	Performance Testing
OWASP ZAP	Security Testing
Docker	Consistent test environment

---

9. Defect Management

- **Tracking Tool:** GitHub Issues / Jira
- **Defect Lifecycle:** New → Assigned → In Progress → Fixed → Retested → Closed
- Severity levels:
  - **Critical:** Blocks major functionality (e.g., login failure).
  - **High:** Breaks core features but has a workaround.
  - **Medium:** UI or minor functionality issue.
  - **Low:** Cosmetic or non-impacting issue.

---

10. Entry and Exit Criteria

Entry:

- All features for a sprint are implemented and unit tested.
- Test environment is ready and stable.

Exit:

- 95%+ test cases passed.
- No open critical or high-severity defects.
- UAT signed off.

---

11. Test Deliverables

- Test Plan & Test Strategy Document
- Test Cases and Checklists
- API Test Collection (Postman)
- Automated Test Scripts
- Test Execution Reports
- Defect Reports

---

12. Risks and Mitigation

Risk	Mitigation
Delayed test environment setup	Use Docker Compose for quick setup
Unstable builds during sprints	CI/CD pipeline with smoke test stage
Limited test data coverage	Prepare diverse test data sets upfront
Inconsistent uploads in S3	Use mock S3 in test environment

---

13. Test Reporting

- Daily status reports to stakeholders.
- Automated test results via CI/CD pipeline.
- Summary report after each sprint:
  - Passed, failed, skipped test cases.
  - Defect metrics.

14. Conclusion

This test strategy ensures a **structured, scalable, and automated approach** to verify DMS quality, covering:

- Functional correctness
- Security
- Performance
- User experience

It reduces risk and ensures smooth delivery for production.