# Automation Test Strategy: Incident Management System

## 1. Automation vs. Manual Testing Strategy

We will prioritize automation for high-risk, repetitive tasks while reserving manual testing for high-cognition and edge-case discovery.

| Category | Type | Strategy | Justification |
|---|---|---|---|
| **Automation** | **Smoke & Regression** | **High Priority** | Critical flows like **Login (User-authentication-01)** and **Incident Creation** must be automated to ensure core stability during every build. |
| **Automation** | **API Validation** | **Full Coverage** | Use the existing Postman collection to validate schemas, 404 behaviors, and security headers automatically via Newman. |
| **Automation** | **Security (Input)** | **Targeted** | Automate Test Case **User-authentication-05** (SQL/Command Injection) to prevent vulnerabilities from reaching production. |
| **Manual** | **Exploratory** | **Ad-hoc** | Necessary for identifying specific rendering issues, such as the **Safari Blank Page (Bug 4 from Bug Report)**. |
| **Manual** | **UX & Accessibility** | **Periodic** | Evaluating if error messages are helpful and if the interface meets accessibility standards (WCAG). |

## 2. Test Pyramid Recommendation

To optimize for speed and reliability, we follow a pyramid structure that favors a heavy base of fast unit tests.

- **Unit Tests (60%):** Focus on Vue components and Node.js business logic.
- **Integration/API Tests (30%):** Validating the communication between services (using your Postman Collection).
- **E2E Tests (10%):** Validating the "Happy Path" user journeys across the full stack.

## 3. Tool Selection (Node.js + Vue.js Stack)

| Tool | Purpose | Pros | Cons |
|---|---|---|---|
| **Vitest** | Unit/Component | Native Vite support; extremely fast; seamless Vue 3 integration. | Newer ecosystem compared to Jest. |
| **Playwright** | E2E Testing | Supports **Safari (WebKit)** rendering; handles async UI perfectly. | Slightly more complex setup than Cypress. |
| **Newman** | API Testing | Runs your **Postman JSON collection** directly in CI/CD pipelines. | Scripting is limited to Postman's sandbox. |

# 4. CI/CD Integration Proposal

We will utilize **GitHub Actions** to enforce quality gates on every Pull Request.

### Example Workflow (.github/workflows/qa.yml)

YAML

```yaml
name: QA Pipeline
on: [push, pull_request]

jobs:
  quality-gate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '20'

      - name: Install Dependencies
        run: npm ci

      - name: Static Analysis (Linting)
        run: npm run lint

      - name: Unit Tests
        run: npm run test:unit

      - name: API Tests (Newman)
        run: |
          npm install -g newman
          newman run "JOES Test.postman_collection.json" --env-var
"baseUrl=${{ secrets.API_URL }}"

      - name: E2E Tests (Playwright)
        run: npx playwright test
```

# 5. Test Data Management Strategy

- **Seeding:** Utilize the credentials provided in your test plan (Admin, Security Analyst, etc.) as a "Golden Dataset" seeded into the test database before execution.
- **Isolation:** Use unique naming conventions for created incidents (e.g., test-incident-${UUID}) to allow parallel test execution without data collisions.
- **Cleanup:** Implement an afterAll hook in the test suite to delete any data created during the session, ensuring the environment remains idempotent.

## 6. Security Testing Integration

To address issues like the **Stack Trace Exposure (Bug 1 from bug report)**, security must be baked into the pipeline.

- **SAST (Static Analysis):** Use npm audit and Snyk to scan for vulnerable dependencies in the package.json.
- **DAST (Dynamic Analysis):** Integrate OWASP ZAP baseline scans against the Swagger documentation endpoint to detect information leakage.
- **API Security:** The Postman collection already includes a test for no stack trace leaked. This will run on every build to prevent regressions.