WAXFEED

E2E Test Suite Summary Report

Comprehensive Testing Analysis & Key Learnings

2150+	68	29	39
TOTAL TESTS	TEST FILES	PAGES	API ENDPOINTS

Generated January 2026 • Playwright Framework

[redacted]

SECTION 01

Executive Summary

This report summarizes the comprehensive end-to-end testing effort for the WAXFEED music discovery platform. The test suite provides extensive coverage across all application routes, user flows, and edge cases.

2150+	68	29	39
TOTAL TESTS	TEST FILES	PAGES COVERED	API ENDPOINTS

Testing Scope

- Core page functionality and navigation flows
- User authentication and protected routes
- Responsive design across mobile, tablet, and desktop
- Accessibility compliance (ARIA, keyboard navigation)
- Security testing (XSS, SQL injection prevention)
- Performance and memory leak detection
- Network resilience and error handling
- Concurrent user simulation and race conditions

SECTION 02

Coverage Breakdown

Test coverage organized by application area and functionality type.

Page Coverage

CATEGORY	PAGES	TESTS	STATUS
Core Discovery	TasteID, Trending, Discover, Search	~200	COMPLETE
Detail Pages	Album, Review, Lyrics, List	~180	COMPLETE
User Features	Profile, Settings, Friends, Stats	~150	COMPLETE
Authentication	Login, Signup, Onboarding	~120	COMPLETE
Social Features	Hot Takes, Reviews Feed, Compare	~100	COMPLETE
Notifications	Notifications, Activity Feed	~60	COMPLETE
Admin	Admin Dashboard, Logo Preview	~40	COMPLETE

Test Categories

ACCESSIBILITY	PERFORMANCE	SECURITY
~150	**~120**	**~100**
ARIA, keyboard nav, focus management, screen readers	FCP, DOM size, memory leaks, debouncing	XSS prevention, input sanitization, auth
ERROR HANDLING	MOBILE	NETWORK
~80	**~100**	**~80**
API errors, JS errors, recovery flows	Touch events, viewports, orientation	Offline mode, retries, caching

SECTION 03

Key Learnings & Patterns

Essential testing patterns and techniques discovered during the test suite development.

1. XSS Detection with Dialog Listeners

Detect XSS vulnerabilities by monitoring for unexpected alert/confirm/prompt dialogs. This catches script execution from malicious input without executing harmful code.

```
// XSS Detection Pattern let xssDetected = false page.on('dialog', async
dialog => { xssDetected = true await dialog.dismiss() }) await
page.goto(`/search?q=<script>alert('xss')</script>`)
expect(xssDetected).toBe(false)
```

2. Network Mocking for Resilience Testing

Use page.route() to simulate slow responses, errors, and network failures. This validates graceful degradation and error recovery without affecting real services.

```
// Simulate API Failures await page.route('**/api/**', async route => { await
route.fulfill({ status: 500, body: 'Server error' }) }) // Verify error
handling UI appears
```

3. Memory Leak Detection via DOM Monitoring

Track DOM element count during navigation cycles. Memory leaks often manifest as continuously growing DOM size or uncleared event listeners.

```
// DOM Size Monitoring const initialSize = await page.evaluate(() =>
document.querySelectorAll('*').length ) // Navigate through pages... const
finalSize = await page.evaluate(() => document.querySelectorAll('*').length )
expect(finalSize / initialSize).toBeLessThan(3)
```

4. Concurrent User Simulation

Use multiple browser contexts to simulate independent user sessions. This validates that the app handles concurrent access without race conditions.

```
// Multiple User Contexts const contexts = await Promise.all([
browser.newContext(), browser.newContext(), browser.newContext() ]) const
pages = await Promise.all( contexts.map(ctx ⇒ ctx.newPage()) )
```

5. Viewport Emulation for Responsive Testing

Test responsive layouts by setting viewport dimensions rather than using device presets in describe blocks (which can cause worker conflicts in Playwright).

```
// Responsive Testing Pattern test.beforeEach(async ({ page }) ⇒ { await
page.setViewportSize({ width: 390, height: 844 }) // iPhone 14 Pro })
```

6. Resilience Testing with Exponential Backoff

Verify that the app implements retry logic with exponential backoff by tracking request timestamps during simulated failures.

SECTION 04

Test File Inventory

Complete list of test files organized by category.

Core Pages (10 files)

- tasteid.spec.ts - TasteID page and archetype display
- trending.spec.ts - Billboard 200, hot reviews
- discover.spec.ts - Feature wheel, new releases
- search.spec.ts - Search functionality, tabs
- album.spec.ts - Album details, tracklist, reviews
- review.spec.ts - Review display, replies
- list.spec.ts - List details, albums
- lists-browse.spec.ts - List browsing, filtering
- lyrics.spec.ts - Lyrics page, track info
- profile.spec.ts - User profiles, activity

Authentication (5 files)

- auth.spec.ts - Login/signup forms
- auth-flows.spec.ts - Auth flows, OAuth
- onboarding.spec.ts - User onboarding steps
- taste-setup.spec.ts - Taste profile setup
- settings.spec.ts - User settings

Social Features (7 files)

- hot-takes.spec.ts - Community debates
- hot-take-new.spec.ts - New hot take form
- reviews-feed.spec.ts - Recent reviews feed

- friends.spec.ts - Friends list, social
- notifications.spec.ts - Notification system
- similar-tasters.spec.ts - User matching
- compare.spec.ts - Taste comparison

Quality Assurance (15 files)

- accessibility.spec.ts - ARIA, a11y compliance
- keyboard-navigation.spec.ts - Tab order, focus
- mobile.spec.ts - Touch, viewports
- performance.spec.ts - Load times, metrics
- visual-regression.spec.ts - Theme, styling
- error-handling.spec.ts - Error states
- error-boundary.spec.ts - Component errors
- form-validation.spec.ts - Input validation
- seo.spec.ts - Meta tags, SEO
- network.spec.ts - Network mocking
- resilience.spec.ts - Retry, recovery
- concurrency.spec.ts - Race conditions
- memory-leaks.spec.ts - Memory monitoring
- timezone-dates.spec.ts - Date handling
- print.spec.ts - Print styles

Advanced Testing (8 files)

- api.spec.ts - API endpoint testing
- integration-tests.spec.ts - Multi-page flows
- edge-cases.spec.ts - Browser quirks
- state-persistence.spec.ts - State management

- browser-history.spec.ts - Navigation history
- cookies-consent.spec.ts - GDPR, cookies
- carousel.spec.ts - Image galleries
- realtime.spec.ts - WebSocket, live updates

SECTION 05

Recommendations

Insights and recommendations based on testing analysis.

1 Strong Coverage Areas

Authentication flows, core page loading, and responsive design have excellent coverage. Security testing (XSS, SQL injection) is comprehensive across all input points.

2 Consider Visual Regression Snapshots

Add Playwright screenshot comparisons for critical UI components. This catches unintended visual changes during refactoring or dependency updates.

3 API Contract Testing

Expand API tests to validate response schemas and error formats. Consider adding OpenAPI schema validation for type safety between frontend and backend.

4 Performance Budget Monitoring

Implement performance budgets with Playwright's performance API. Track Core Web Vitals (LCP, FID, CLS) and fail tests if thresholds are exceeded.

5 CI/CD Integration

Run tests in parallel across multiple workers in CI. Use sharding for faster execution and generate HTML reports for each build.

SECTION 06

Best Practices Established

Testing patterns that should be followed for future development.

Test Structure

- Group tests by feature in describe blocks
- Use descriptive test names that explain the expected behavior
- Keep tests independent - each test should set up its own state
- Use fixtures for common setup (authentication, mock data)

Async Handling

- Use waitForSelector or waitForTimeout appropriately
- Prefer explicit waits over arbitrary timeouts when possible
- Handle race conditions with proper synchronization

Error Prevention

- Check element existence before interaction with count()
- Use try/catch for operations that may fail gracefully
- Verify page state before asserting on dynamic content

Maintainability

- Extract common selectors to constants or page objects
- Use data-testid attributes for stable selectors
- Keep test files focused on single features