Product Requirements Document (PRD)

Broken Link & Image Scanner

1. Overview

- **Product Name:** Broken Link & Image Scanner
- **Primary Platform:** Google Chrome (with plans to expand to Firefox and Edge)
- **Purpose:** Detect broken or dead links and missing images on WordPress pages (and potentially any website), helping users quickly identify and resolve SEO and user-experience issues.

2. Objectives

1. **Identify Broken Links & Images Quickly:** Provide an easy-to-use tool that scans the current webpage for dead links (HTTP status 404, 403, etc.) and missing images.
2. **Optimize SEO & User Experience:** By detecting these issues, site owners or editors can correct problems and improve page rankings and user satisfaction.
3. **Extendibility & Scalability:** Allow for future expansions (e.g., deeper SEO checks, scanning multiple pages, export of reports, etc.) and cross-browser functionality.

3. Key Features

1. **Link Scanner:**
    - Scans all `<a>` tags for their `href` attribute and tests for response codes.
    - Flags any link that returns an error (404, 403, 500, etc.).
    - Allows the user to see the broken links in a report or summary.
2. **Image Scanner:**
    - Scans all `<img>` tags for valid `src` attributes.
    - Checks if any images fail to load or return an HTTP error (e.g., 404).
    - Displays a list of all images that failed to load.

3. **On-Page Report:**
    - Summarizes all broken links and unloaded images in a popup or a side panel.
    - Shows the total count, plus specific URLs/paths causing errors.
4. **User-Friendly Interface:**
    - Includes a simple icon in the browser toolbar.
    - Clicking the icon opens a popup or dedicated page with results, including:
        - List of broken links
        - List of broken/unloaded images
        - Status codes
        - Suggestions for how to fix them
5. **Optional Detailed View / Export:**
    - Allows the user to export the findings as a JSON or CSV file.
    - Includes the option to re-check selected items on demand.

4. Functional Requirements

4.1 Browser Extension Structure (Chrome, Manifest V3)

1. **Manifest File:**
    - Permissions required: `activeTab`, `scripting`, possibly `storage`.
    - Uses the Manifest V3 format.
2. **Background Service Worker (if needed):**
    - Listens for messages from the content script to manage scanning results.
    - Performs any network requests that require background execution (if not handled directly in the content script).
3. **Content Script:**
    - Injected into the page to traverse the DOM and identify all links and images.
    - Sends requests to each link/image to verify its status code or checks if the image is loaded.
    - Passes data back to the service worker or directly to the extension's

popup script.

4. **Popup (UI) / Extension Page:**
    - Displays scanning results: number of total links, broken links & their URLs, unloaded images & their URLs.
    - Option to re-scan or export results.

4.2 Core Scanning Logic

1. **DOM Parsing:**
    - Script finds all `<a>` and `<img>` tags on the active page.
    - Collects their URLs (`href` or `src`).
2. **HTTP Status Checks:**
    - For each link/image, perform a `fetch` request to see if the response is valid (status < 400).
    - For images, check the load event or do the same `fetch` approach.
    - Store the results in an array of “valid” vs. “broken” items.
3. **Reporting:**
    - Return the array of broken link/image data (URL, status code, short description of the issue).

4.3 User Interaction / UI Flows

1. **Extension Icon Clicked:**
    - Inject or trigger the scan on the current tab, if not already scanned.
    - Show a popup or new tab with a quick summary of results.
2. **View Detailed Report:**
    - If user wants more info, allow either a clickable item in the popup that expands a list of all issues, or a link to an options page with full details.

4.4 Performance & Constraints

- The extension must handle pages with many links without hanging or crashing.
- Must manage asynchronous requests efficiently (e.g., using promises, concurrency limits).

5. Non-Functional Requirements

1. **Performance:**

- Scanning should complete within a few seconds for typical pages (< 500 links).
- For larger pages, handle concurrency gracefully and possibly show a progress indicator.

2. **User Experience:**
   - Minimal clicks needed to scan the page.
   - Clear, concise reporting of issues.
   - No cluttered UI or extraneous features in the initial version.
3. **Security & Privacy:**
   - Only request permissions strictly needed for scanning links/images.
   - Adhere to browser extension security guidelines (Manifest V3 policies).
4. **Compatibility:**
   - Primary focus on Chrome first.
   - Architecture that can be ported to Firefox (using `browser` APIs or polyfills) and Microsoft Edge with minimal changes.
5. **Scalability:**
   - Codebase structured in a way to easily add features like scanning multiple pages or entire sites later.
   - Potential for centralized logging if the user chooses to link an account in the future.

6. Technical Approach

1. **Manifest V3-based Implementation:**
   - Use a service worker instead of a background page.
   - `scripting` API to inject content scripts.
2. **Content Script Implementation:**
   - Query DOM for `<a>` and `<img>` tags.
   - Use `fetch` to confirm link validity.
   - Check load events for images or do parallel `fetch` checks.
3. **Data Storage:**
   - In the initial version, results can be displayed immediately without

storage.

- Optionally store in `chrome.storage.local` for future reference.

4. **UI / Popup:**
   - Built with HTML, CSS, JavaScript (or a lightweight framework if desired).
   - Real-time updates from the content script via message passing.

7. User Flow Diagram (High-Level)

1. User clicks extension icon.
2. Extension injects or runs content script.
3. Script scans the DOM.
4. Script returns data to the extension.
5. Extension displays summary in popup.
6. User can expand details or export results.

8. Project Milestones & Timeline

1. **Milestone 1: Basic Scanning Logic (2-3 days)**
   - Implement content script to find links/images and fetch status codes.
   - Console-log results for broken links/images.
2. **Milestone 2: UI & Reporting (3-5 days)**
   - Create extension popup interface.
   - Display broken link/image data in a readable list.
3. **Milestone 3: Polishing & Testing (1-2 weeks)**
   - Test across multiple websites, including WordPress-based and non-WordPress-based.
   - Optimize performance for large pages.
   - Handle edge cases (e.g., relative links, JavaScript-based links).
4. **Milestone 4: Manifest V3 Packaging & Deployment (1 week)**
   - Prepare final extension package for Chrome Web Store submission.
   - Complete any required compliance (privacy policy, etc.).

5. **Future Milestones (beyond initial release):**
    - **Firefox / Edge Port:** Adapt manifest if necessary.
    - **Mobile App:** Investigate feasibility with a separate codebase or frameworks like React Native.
    - **Additional SEO Features:** Extend scanning to check meta tags, page speed metrics, etc.

9. Risks & Dependencies

- **Changing Web Standards:** Ongoing changes to Chrome extension API (Manifest V3).
- **Cross-Browser Inconsistencies:** Different extension APIs for Firefox and Edge.
- **Network Restrictions:** Some sites may block requests or yield inconsistent status codes if scanning is too aggressive.
- **Performance on Very Large Pages:** Need efficient queueing of fetch requests to avoid browser freezing.

10. Acceptance Criteria

- **Core Features Work:** Scans current active tab, identifies broken links/images.
- **Clear Reporting:** Summarizes results in a well-formatted UI.
- **Performance:** Completes scanning within a reasonable time on typical sites.
- **Stable Behavior:** No crashes, memory leaks, or major UI bugs during usage.
- **Manifest V3 Compliance:** Successfully published (or ready to publish) in the Chrome Web Store.

End of PRD — Broken Link & Image Scanner