



# AnZimmerman ZIM Reader – PRD & Architecture Document

## 1. Product Requirements Document (PRD)

### Feature Scope

- **Core Offline Reader:** Ability to open and display content from ZIM files (offline Wikipedia and similar archives) with full fidelity (text, images, links, styles) <sup>1</sup>. Content should render in a reader view (likely a webview or HTML renderer) and allow basic navigation (scroll, next/prev, table of contents if available).
- **Library Management:** Manage a library of multiple ZIM files. Users can add/remove ZIM archives and see metadata (title, size, description). Support listing available archives and selecting one to browse. Optionally integrate a **ZIM downloader** to fetch new archives from an online catalog.
- **Search:** Full-text search within a ZIM's content (article titles and possibly article body) <sup>2</sup> <sup>3</sup>. Provide a search UI for queries with results listing article titles (and snippets if possible). Leverage ZIM's built-in title index or full-text index if present <sup>3</sup>, otherwise generate an index on the fly. Must handle large archives efficiently.
- **Semantic Mesh (Knowledge Graph):** Visualize relationships between articles in a “semantic network” or concept map. Each article is a node, and edges represent links or semantic similarity (e.g. “See also” links connecting topics) <sup>4</sup>. This interactive graph allows users to explore related content and discover articles by navigating the network of connections. The mesh view should highlight the current article and its neighbors, with options to expand or focus on different nodes.
- **Bookmarks & History:** Users can bookmark articles for later reference and view a history of recently read articles <sup>2</sup>. The system should record visited pages (with timestamps) and allow clearing history. Bookmarked pages should be saved persistently (per ZIM or globally) and listed in a bookmarks section.
- **Navigation & Misc:** Basic navigation features like back/forward within reading sessions, jumping to the main page of the ZIM, and possibly an interactive table of contents if the ZIM has one. Additionally, if the ZIM archive supports multiple languages or contains category info, provide ways to filter or jump by those categories (stretch goal).

### Target Platforms

- **Desktop (Windows/Linux/macOS):** A cross-platform desktop application, built with Tauri (Rust backend, HTML/JS frontend) for native file access and performance. The desktop app provides the full feature set – library management, reader, search, etc – in a unified interface. Tauri is used to package the app for each OS, leveraging the same React/Tailwind UI.
- **Web (Browser/Extension):** A web application or browser extension for users who prefer in-browser usage. For example, the Chrome extension “Zimmer” is used to save and view pages offline <sup>5</sup>. The web version runs entirely client-side using the ZIM JS library (no server required) and can open ZIM files provided by the user (via file picker or drag-drop). It may have a reduced feature set due to browser limitations (e.g. no direct filesystem access), but core reading and

search should work in memory or via IndexedDB. The Chrome extension variant can integrate with browser features (context menus for saving pages, etc.) <sup>6</sup>.

- **Mobile (Android/iOS):** A mobile app for on-the-go access. On Android, leverage Tauri Mobile (Android WebView + Rust) to re-use the core codebase <sup>7</sup>. The mobile UI will be adapted for touch input and small screens (larger touch targets, swipe gestures, bottom navigation bar) <sup>8</sup>. iOS support could be via a similar approach (once Tauri supports iOS) or a React Native/WebView wrapper. The mobile app focuses on efficient offline storage (handling large files on device) and a simplified UI for one-handed use.

## UX Principles and Themes

- **Content-Focused Experience:** Emphasize a clean reading experience with minimal distractions. The UI should be simple and decluttered when viewing articles – content takes center stage (akin to a reader mode). Navigation chrome (toolbars, buttons) will be unobtrusive, possibly auto-hiding or kept to edges, to maximize content space.
- **Consistency & Theming:** Maintain a consistent design language across platforms. Use a unified design system with Tailwind CSS utility classes and CSS variables for theming. Support multiple visual themes (e.g. a palette of 9 themes as used in the Chrome extension) <sup>9</sup>, including light/dark modes and a system default mode switch. Users can toggle dark mode and select theme preferences on all platforms for comfort.
- **Responsiveness:** Design layouts that adapt to various screen sizes gracefully (desktop widescreen, tablet, phone). Use responsive design techniques (flexbox/grid and relative units via Tailwind) to ensure the library and reader views reflow appropriately (e.g. grid of cards on desktop, single column list on mobile).
- **Fast and Offline-First:** Optimize for quick load times and smooth interactions, even on low-end devices or without internet <sup>10</sup>. The app should launch quickly and open articles within a second or two, giving the feel of a native encyclopedia. All features (search, navigation) should work offline once content is loaded, with no dependency on external services.
- **Accessibility:** Adhere to accessibility best practices so that offline knowledge is available to all users. Ensure high contrast themes are available, support screen readers (proper semantic HTML for content), and keyboard navigation (especially on desktop) <sup>11</sup>. This includes accessible focus states and ARIA labels for interactive controls in the UI.

## Technical Constraints

- **Clean-Room Implementation:** All ZIM file handling is done with clean-room libraries based on the openZIM specification <sup>1</sup>. No code from existing GPL or non-permissive implementations is used, to ensure the project remains MIT licensed. The system relies on the custom **zimlib** modules (in Go, TS, Python, etc. as needed) built from the spec <sup>12</sup>. This guarantees full control over features and avoids licensing issues (the entire codebase is MIT licensed <sup>13</sup>).
- **File Format and Performance:** ZIM files can be very large (many gigabytes, millions of entries), so the architecture must handle large file I/O efficiently. Use streaming and lazy loading – e.g. do not load entire file to memory. The ZIM reader library should read only needed blocks (clusters) on demand. The core library supports random access to articles via an index and decompression of clusters on the fly <sup>14</sup>. It must manage memory carefully (use of memory-mapped files or sequential reading) to work on low-end hardware.
- **Platform-Specific Constraints:** In web contexts, the app cannot use OS file paths or spawn native threads. Therefore, the browser version should operate on an **ArrayBuffer** of the ZIM (possibly loaded via the File API) <sup>15</sup>. In contrast, the desktop/mobile (Tauri) version can use native file APIs and multi-threaded Rust code for heavy tasks (like indexing). We will maintain an abstraction layer so that high-level code (search, fetch article) calls either the JS library (in browser) or Rust backend (in Tauri) depending on platform.

- **Security & Sandboxing:** The reader will display HTML content from ZIM archives. To avoid security issues (especially if ZIMs contain script or malicious content), the app should sanitize or securely sandbox the content. In a Tauri/desktop context, use an embedded browser view with appropriate CSP (Content Security Policy) and no remote network access for that content. In a web app, serve content in a sandboxed iframe or sanitize the HTML to strip scripts. This ensures reading offline content cannot compromise the user's system.
- **MIT License Compliance:** All third-party libraries or data used must be MIT or similarly permissive. If we integrate any code generators or AI builder outputs, those outputs must be reviewed for license compatibility. No GPL or proprietary code can be included. This extends to any design assets – use open or custom icons (the project uses Lucide icons which are open-source <sup>16</sup>, for example) and ensure no dependency with incompatible license is pulled in.

## Target Audience / User Personas

- **Student/Researcher (Offline Learner):** A student or educator in a low-connectivity environment (e.g. rural area, developing country) who needs access to knowledge offline. They download Wikipedia (or other educational content) as ZIM files and use the app to read articles, search for topics, and save bookmarks for study. Success for this user is the ability to quickly find relevant information without internet. (This aligns with the core mission of Kiwix – targeting users on low-end devices in the Global South for offline knowledge <sup>10</sup>.)
- **Traveler (On-the-go Reader):** A traveler with limited or expensive internet who wants an offline library of reference (e.g. Wikivoyage, Wikipedia). They use the mobile app to lookup information on the go. Important for them is that the app is lightweight (runs on a mid-range phone), and the UI is intuitive even on a small screen. They value features like search and history to re-find info while offline.
- **Knowledge Explorer (Power User):** A tech-savvy user or researcher who not only wants to read articles but also explore connections between topics. This persona is excited about the **semantic mesh** feature – they use the app to discover related articles through the visual graph, turning the encyclopedia into an interactive knowledge map. They might use the desktop app for easier viewing of the semantic graph and possibly export parts of the graph for presentations or research.
- **Archivist/Data Hoarder:** A user who downloads large collections (perhaps entire Wikipedia dumps) for archiving purposes. They use the app to verify and skim content. For them, performance and stability with huge archives is key. They might also use the extension to scrape specific web pages into custom ZIMs. This persona cares that the app can handle multi-GB files, and that it provides organizational features (folders, tags for different archives) and possibly ways to export content if needed.

## Success Metrics

- **Adoption & Usage:** Track the number of active users and downloaded archives. For example, number of ZIM files opened per user, and frequency of usage (daily active users opening the app). If the target audience includes offline communities, success could be measured by deployments (e.g. installs in schools or libraries in target regions) and user feedback from those deployments.
- **Content Access Performance:** Quantitative metrics like **time to first article load** (after opening a ZIM) and search query performance. Aim for < 2 seconds to open an article by title, and fast incremental search (< 0.5s for suggestions, < 2s for full-text results on a large archive). These can be measured in testing with large ZIMs. Another metric: memory footprint while a large ZIM is loaded – ensuring it can run on devices with limited RAM.
- **User Satisfaction & Retention:** Qualitative feedback and ratings. High satisfaction if users find it easy to use and reliable. Track bookmark usage and semantic mesh usage – e.g. if a significant

portion of users use the semantic graph to navigate (indicating the feature's value). Also measure how many articles on average a user reads per session (higher could indicate the app facilitates deep dives into content).

- **Successful Cross-Platform Sync:** If applicable, measure how seamlessly users transition between devices. For instance, if we add a cloud sync for bookmarks or history in future, success might be users taking advantage of reading on desktop and mobile interchangeably. (Initial version may not have sync, but local continuity is a metric – e.g. user opens app and can pick up where they left off).
  - **Development Velocity (Builder Agent Workflow):** Since the project leverages AI builder agents for implementation, one internal success metric is the speed and accuracy of module development. For example, the ability to generate a functional UI component via an agent in a single iteration, or reduction in bugs and rework needed per feature. This can be measured by number of tasks completed by agents vs manually, and the average iterations per task. (This is a meta-metric relevant to the collaborative builder-agent process.)
- 

## 2. Architectural Overview

### High-Level System Components

- **Frontend UI (React + TypeScript):** The user interface is built as a React application (with TypeScript) for modularity and reuse. This includes all views (library, reader, search, etc.) and components (buttons, lists, graph visualization). Styling is done with Tailwind CSS for consistency. The frontend handles user interactions, presents content, and dispatches actions (like opening an article or performing a search).
- **ZIM Parsing Engine:** The core logic to read and write ZIM files, abstracted as a library module. In a web context, this is the **ZIMReaderBrowser** (a JS/TS library that reads from an ArrayBuffer in memory) <sup>17</sup> <sup>15</sup>. In the desktop app, this might be a native Rust module or a binding to the same logic (to leverage native file I/O). The parsing engine provides APIs to: open a ZIM archive, fetch an article by URL or index, stream article content (decompressing cluster data), and perform searches. It implements the ZIM spec: reading the header, directory entries, cluster blobs, etc. All platform variants call this layer for data operations.
- **Search Index & Semantic Engine:** A component responsible for enabling fast search and semantic analysis. It may consist of:
  - A **Search Indexer** that, upon loading a ZIM, builds an index (if none provided) of article titles and possibly full text. This could be an in-memory index (for smaller ZIMs) or use a lightweight database/search library on disk for large ones. On web, a simplistic approach (scanning titles or using a WASM full-text search library) is used, whereas on desktop, a more robust index (e.g. an embedded full-text search engine or Rust crate) could be employed for better performance.
  - A **Semantic Analyzer** that constructs the “semantic mesh”. This can parse link information in the ZIM (e.g. find all hyperlinks between articles) to build a graph of connections. It might also use natural language processing to find related topics (e.g. generating embeddings for articles and computing similarity, potentially a future enhancement). The output is a graph data structure: nodes (articles) and edges (relationships). This is used by the Semantic Mesh UI to render the network.
- **Backend Services (optional):** While the application is primarily offline, certain components might be structured as services or background workers. For instance, in the Tauri app, heavy tasks (like generating a search index or reading large binary chunks) can run in a Rust thread or command invoked via Tauri’s message system. The Chrome extension uses a background script (Service Worker) for tasks like page capture and archive writing <sup>6</sup>. If a user opts to download

ZIM files from the internet, a small networking module/service will handle those downloads (e.g. with progress, pause/resume).

- **Data Storage:** The app needs to store some data outside of ZIM files: user bookmarks, history, and settings. On desktop, this could be a local JSON or SQLite database accessed via the backend (Rust) or via embedded IndexedDB in the frontend. On web (extension or PWA), use `localStorage` or IndexedDB for persistence of bookmarks/history. This storage module exposes a consistent interface to the UI for reading/writing user data. For example, a `BookmarkManager` service that the UI can call to add or fetch bookmarks, regardless of platform.

## Data Models

- **ZIM File Data Structures:** The system represents ZIM content using internal data models reflecting the file format. Key models include:
- **ZimHeader:** High-level metadata from the file header (magic number, version, counts of entries/articles/clusters, etc.) <sup>18</sup>. For example, the header tells how many articles and clusters are in the file and where index tables are located.
- **Article Entry:** Represents a directory entry for a content page. Fields include an index (offset in the directory), namespace (e.g. `A` for article, `I` for image) <sup>19</sup>, the article's URL/path, its title, the cluster number and blob index where the content resides, and MIME type info <sup>20</sup>. In memory, we might model this as an object or class `ZimEntry` with those properties. We also flag if an entry is a redirect or an article.
- **Redirect Entry:** A special directory entry that points to another entry (via a redirect index) instead of containing content <sup>21</sup>. The model for this holds the redirect target index and essentially acts as an alias. The UI or engine will seamlessly follow redirects when retrieving content.
- **Cluster and Blob:** The raw content in a ZIM is stored in clusters of compressed data. Each cluster contains one or more "blobs" which are the binary content of articles (HTML text, images, etc.). The engine handles clusters transparently: given an Article Entry (with cluster number and blob index), it locates the cluster file offset via the cluster pointer table, decompresses the cluster (if compressed), and then extracts the specific blob for the article <sup>14</sup>. We typically don't expose cluster/blobs as UI-level models, but the concept is present in the architecture. Performance considerations like cluster size and compression type (Zstandard, etc.) are handled at this layer.
- **ZimArchive Model:** An object representing an open ZIM file, which might include the header, a list (or lazy loader) of all entries, and methods to get content or search. This acts as a high-level data model that the application logic uses. For example, `ZimArchive.getArticle(url)` to retrieve an article's content, or `ZimArchive.search(query)` to return a list of Article Entries matching a query.
- **Semantic Graph Model:** Data structures for the semantic mesh. Possibly define a `GraphNode` (with references to a ZIM article entry plus maybe computed metadata like importance or topic categories) and `GraphEdge` (link between two nodes with a type: hyperlink, category link, similar content, etc.). The graph might be computed on the fly for a given context (e.g. centered on the current article's neighborhood) to avoid having to load the entire wiki graph. This model will be used by the Semantic Mesh view to render interactive nodes. It could also include spatial coordinates if we pre-compute a layout for display, or that can be left to the front-end visualization library.
- **User Data Models:** Simple models for user-specific data:
- **Bookmark:** Contains at least an article reference (which ZIM and which article) and maybe a timestamp or folder/tag.
- **History Entry:** Article reference plus timestamp of when viewed.

- These could be just stored as records in a local database or JSON, but we define them so that multiple modules (reader UI, history panel, etc.) use a consistent format. E.g., `Bookmark = { zimId, articleIndex, title, addedAt }`.
- **UI State Models:** The application state can be represented with models such as:
- **ZimList / Library:** Array of available ZIM archives (each with id, name, maybe an icon or description).
- **AppState:** An overall structure containing things like `currentZimId`, `currentArticle` (maybe store as article entry or just index), `searchQuery`, `searchResults[]`, `viewMode` (if we have different modes like library vs reader vs semantic mesh). This helps in coordinating state (especially when using a global store).
- We will keep these models simple (primarily plain objects or TypeScript interfaces) so that they can be easily shared across web and backend if needed (for instance, the Rust backend could mirror some structures when sending data to the UI).

## Key Workflows & Flow Diagrams

- **Content Loading Flow:** When a user opens an article, the system follows a sequence: **Select Article -> Engine Fetch -> Display**. In detail: the UI triggers an action (e.g., user clicks a link or search result) with an article identifier. The engine looks up the Article Entry in the directory (if we have the index or path, it finds the matching entry). From the entry, it retrieves the cluster number and blob index, then finds the cluster's file offset via the ZIM header's cluster pointers, reads and decompresses that cluster, and extracts the article HTML <sup>14</sup>. The HTML content is then returned to the UI. The UI's Reader component takes this content and renders it in the webview or content area (ensuring any necessary base URL or image loading mechanisms are handled – images might be blobs that also need fetching from the ZIM through the engine). This flow is optimized via caching: recently accessed clusters could be cached in memory to avoid repeated disk reads if multiple articles in the same cluster are read sequentially.
- **Search Query Flow:** The user enters a query in the search bar, initiating a search workflow. If a full-text search index is available in the ZIM (some ZIMs include a precomputed index), the engine will utilize it <sup>3</sup>. Otherwise, on first search, the engine may build an index: e.g., iterate through all article titles (and maybe content) to create a list or inverted index in memory. On subsequent searches, it queries this index. The flow: **User Query -> Normalize Query -> Query Index -> Rank Results -> Return Results**. Results are a list of matching Article Entries (with maybe a snippet or highlight of context). The UI receives these results and displays them in a Search Results view, sorted by relevance. Clicking a result then goes through the content loading flow. For large archives, if building a full index is too slow or memory-heavy, an alternative flow is incremental search: e.g. search through titles first (which can be done quickly via the title index in the ZIM), display those results, and optionally kick off a background thread to search full text in the archive, updating results when ready.
- **Semantic Mesh Generation Flow:** When a user opens the semantic mesh view (likely from an article page by choosing "Explore Links" or similar), the system generates the local graph around the current context. **Current Article -> Gather Related Articles -> Build Graph -> Visualize**. To gather related nodes, the engine might fetch all outgoing links from the current article (parsing the HTML for anchor tags pointing to other wiki articles) and also find incoming links (articles that link to this one, if we have that data or can derive it by scanning a subset of articles or using a precomputed link index). It may also use category metadata (if ZIM includes category pages) or an algorithm to find "similar" content (e.g. articles sharing many links or keywords). Once the related nodes (and possibly second-degree connections among them) are identified, a Graph structure is assembled in memory. The UI then renders this via a graph visualization library or custom canvas/SVG: nodes as circles or icons, connected by lines (edges). The user can click on

any node to open that article (triggering content load flow) or expand that node to see further connections (which would loop back to rebuild the graph for a new focus).

- **Inter-Module Communication:** The app is structured so that front-end components communicate with the engine and backend through defined interfaces. For instance, in the Tauri app, when the React UI needs article content, it calls a Tauri command (exposed via Rust) like `getArticle(zimId, articlePath)`; the Rust backend uses the ZIM parsing library to retrieve the data and returns it. Similarly, for search: `searchArticles(zimId, query)` triggers a Rust function which may spawn a thread to search and then sends results back to the UI. This asynchronous message-passing keeps the UI responsive. In a pure web context, since there is no separate process, the communication is function calls to the JS ZIM library (which might run in a Web Worker if needed to keep UI smooth). Thus, **in-browser** modules communicate via direct function calls or postMessage to workers, whereas **in Tauri** the front-end and back-end communicate via IPC provided by Tauri (under the hood, message passing between the webview and Rust). We maintain consistent API shape for these calls so that the UI code doesn't drastically change per platform. For example, a higher-order `ContentService` in the front-end may abstract whether it's calling a web worker or a Tauri backend – providing a unified promise-based API to the React components.
- **Browser vs Tauri Differences:** One notable inter-module difference is file access. In Tauri, the backend can directly open file paths (e.g. using Rust std::fs to open the .zim), whereas in a browser, we must ask the user to upload a file which we then read into memory. To bridge this, the library in browser can accept an `ArrayBuffer` of the entire file (which might require the user to have enough memory). Some mitigation: the Chrome extension can use the Chrome storage API or filesystem API to store the file, and feed data on demand, but it's still essentially reading from an in-memory or blob source. In both cases, once the file is "opened" in our app, subsequent operations (get article, search) treat it similarly through the ZIM reader interface. Another difference is multi-threading: in Rust we can use threads for e.g. building the search index; in browser we rely on web workers (which are single-threaded but concurrent). Our architecture accounts for these differences by encapsulating platform-specific logic in modules (e.g., a `IndexBuilder` that uses Rust Rayon threads on desktop vs uses a simplified approach or yields periodically in JS on web).

## Component Diagram

*(High-level depiction of modules and their interactions, for reference)*

22 23

The architecture is composed of core **zimlib** libraries in multiple languages (Python, TS, Go, PHP as per project scope), all built against the common ZIM specification <sup>24</sup>. On top of these libraries, we have **tools** like the Rust/Tauri application (for downloading and reading ZIMs) and various language-specific APIs (Python FastAPI, Node/Express, etc.) for accessing ZIM functionality via services <sup>25</sup>. The focus for the reader app is the Tauri/React tool and the Chrome extension front-end, which both utilize the TypeScript `zimlib.ts` for core functionality in-browser and potentially a Rust or Go backend in the desktop app for heavy lifting.

*(Note: The diagram referenced above shows multiple language libraries all feeding into the common spec and being used by different interfaces. In our context, the TypeScript library plus the Rust Tauri backend are most relevant.)*

### 3. UI/UX Flow and Wireframe Outline

#### Page / View Hierarchy

The application UI is organized into a set of core views, accessible via a simple navigation structure (tabs or sidebar on desktop, a hamburger menu or bottom tabs on mobile): 1. **Library View (Home)**: Shows the list of available ZIM archives and possibly options to add/download new ones. 2. **Reader View (Article)**: Displays the content of a selected article. This is the main reading interface. 3. **Search Results View**: Shows search input and results list. (On mobile this might be a separate screen; on desktop it could be a panel or modal overlay on the Reader). 4. **Semantic Mesh View**: An exploratory interface for the knowledge graph of the current archive or article context. 5. **Bookmarks & History Views**: Lists of saved bookmarks and recent articles. These could be separate tabs or combined with the Library or Settings. 6. **Settings/About**: (If needed) for theme selection, preferences, and about info. Possibly included as a section or modal.

**Navigation Structure:** On desktop, a top menu or sidebar could provide access to Library, Bookmarks, Settings, etc., while the reader itself might be fullscreen with an overlay toolbar. On mobile, use a bottom navigation bar or a drawer menu for switching between views (Library, Bookmarks, Settings), and within an open archive, use in-page UI to access search or semantic view.

#### Library View (ZIM Library/Home)

- **Layout:** likely a grid or list of ZIM archive “cards”. Each card shows the archive’s name, an icon or cover image (if available, e.g., a Wikipedia globe or custom icon), size, and possibly a short description. If the archive is not yet loaded (e.g., available to download), it might show a download button or progress.
- **Key UI Elements:**
  - Header: Title (“Library” or app name), and a button to add archives (e.g., “+ Add ZIM” which opens a file picker or triggers download from catalog).
  - ZIM Cards: each card is a clickable component containing the archive title, maybe number of articles, etc. Clicking it opens that archive (navigates to Reader or some “browse” view inside the archive). For example, a “Wikipedia English” ZIM card opens the main page of that archive.
  - Optionally, a search bar at the top to filter archives by name (useful if many archives).
- **Interactions:**
  - Clicking an archive card enters that archive (transition to Reader view showing the main page of that ZIM).
  - The “Add” button triggers an action (desktop: open file dialog to import a .zim file from disk; or open a list of known ZIMs to download).
  - Right-click or long-press (mobile) on a card might open options (e.g., Remove archive, View details).
- **State Indications:** If an archive is currently open (in background) or recently opened, it might be highlighted or show a “Resume reading” action if we want to jump back in. Downloading archives will show progress on the card (with a progress bar overlay).

#### Reader View (Article Display)

- **Layout:** A top bar and content area.
- Top Bar: contains navigation controls (Back, Forward, maybe an “Open Library” home button), a title of the article, and quick actions like **Search (magnifying glass)**, **Bookmark (star icon)**, and a **Menu** (for additional options). On mobile, the top bar might be slim or auto-hide as you scroll.

- Content Area: the main area showing article content (HTML). This likely uses an embedded web view or a scrollable `div` with the rendered HTML. It should apply the selected theme (for example, dark mode – ensure the article text/background follow the theme).
- If the article has images, they should be displayed (the engine provides a way to retrieve images from the ZIM, possibly via blob URLs or base64 data).
- **Key UI Elements:**
- **Article Content:** The styled text of the article, including headings, paragraphs, images. We will inject some CSS (perhaps from the ZIM if it contains a stylesheet in the `S` namespace, or our own default wiki style) so that it looks nice. Links within the article are clickable – clicking an internal link loads that new article (staying in Reader view, updating the content and title accordingly).
- **Back/Forward Navigation:** Users can go back to previously read articles (history stack) or forward if they went back. This works akin to a web browser navigation.
- **Bookmark Toggle:** A star (or bookmark icon) that can be toggled to add/remove the current page from bookmarks. When a page is bookmarked, the icon appears filled.
- **Search Icon:** Opens the Search view (perhaps as an overlay or navigates to a new view) to search within the current archive.
- **Semantic Mesh Icon:** An icon (maybe a network graph symbol) that opens the semantic mesh view for the current article. This might be in a menu or directly visible if space permits.
- **Overflow Menu:** Additional options: e.g., “Open in browser” (if online copy exists), “Copy link”, “Share” (on mobile share the title/text), or “Article info” (metadata, if any).
- **Interactions:**
  - Scrolling through content (with maybe a scrollbar indicator).
  - Pinch-to-zoom on mobile for images or text (or a font size adjustment in settings).
  - Tapping links to navigate.
  - Long-pressing text might allow copy (we should ensure text selection works for copy/paste).
  - Pull-to-refresh gesture might be repurposed to quickly scroll to top (since content is static offline).
  - If the user opens the semantic graph, that likely takes over the view (or splits the view) to show related topics.
- **States:**
  - **Loading State:** When an article is being loaded (after click but before content is ready), show a spinner or skeleton content in the reader. Since local access is fast, this state might be brief, but is important for large images or slower devices.
  - **Error State:** If an article fails to load (corrupt file or missing entry), show an error message in the content area (“Article could not be loaded”) perhaps with a retry option.
  - **Empty State:** If no article is loaded (e.g., user hasn’t opened one yet, or after closing an archive), the reader area can show a placeholder (like an app logo or prompt to select an article).

## Search View

- **Layout:** A search input field at top and a list of results below it.
- On desktop, this might appear as a sidebar or modal overlay on the Reader view (so users can search while an article is open). Alternatively, it could be a separate page that replaces the reader until an article is chosen.
- On mobile, likely a separate screen: tapping search icon from Reader pushes a Search screen where the top is an input and the results populate below.
- **Key UI Elements:**
- **Search Bar:** An input box where user types their query. It should autofocus when opened. Include a clear (“x”) button to reset the query. Possibly also allow filtering by namespace (all content vs titles only) with an icon or dropdown.

- **Result Items:** Each result shows an article title (with the query terms highlighted if relevant) and maybe a short snippet or description (e.g., the first sentence of the article). If an article title is very descriptive, snippet can be optional. We also might show the namespace or an icon if results can include images or other content types.
- **No Results Message:** If nothing matches, show a friendly message ("No articles found for 'X'") possibly with suggestion to check spelling.
- **Interactions:**
  - As the user types, we could provide live suggestions or incremental results (if performance allows). This can help them refine query. Or require pressing Enter to fetch results, depending on index speed.
  - Clicking/tapping a result item immediately navigates to that article in the Reader view (closing/hiding the search view). On desktop, if search is overlay, it could simply populate the reader; on mobile, it might navigate back to the reader screen with the new content.
  - If search covers multiple archives (in case user wants to search across all loaded ZIMs), there could be a filter to choose which archive to search. But initially, assume search is within the current open archive.
- **States:**
  - Loading state** for search: if a query takes noticeable time (e.g., first time indexing), show a spinner or "Searching..." indicator.
  - Results state:** Once results loaded, the list appears.
  - Error state:** If search fails (e.g., out of memory for index), show an error or at least log it. Possibly prompt user to narrow query.

## Semantic Mesh View

- **Layout:** An interactive canvas or SVG area taking most of the screen, with a control toolbar.
- The main area will render nodes (circles or bubbles labeled with article titles) and lines between them representing relationships. Likely using a force-directed layout or similar to distribute nodes.
- A small toolbar might be overlaid (top-right or bottom) with controls: e.g., zoom in/out, center on current article, toggle types of links (to filter the view, e.g., show/hide certain relationship types).
- **Key UI Elements:**
  - Graph Nodes:** Visual representations of articles. Possibly color-coded by some category (for instance, the current article is highlighted, direct links are one color, secondary links another). Each node will show the article title (either always, or on hover/tap to avoid clutter; perhaps only the main nodes are labeled and others show tooltip on hover).
  - Graph Edges:** Lines connecting nodes. If multiple types of relations are present, different line styles or colors could indicate them (solid line for direct wiki link, dashed for category, etc.). If the mesh is specifically "semantic" (like related by content similarity), edges might have no explicit distinction aside from weight.
  - Current Focus Highlight:** The node corresponding to the article from which the user opened this view is emphasized (e.g., larger or with a glow). This helps orient the user about context.
- **Toolbar/Controls:**
  - A **Center/Home button** to recenter the graph on the original article node.
  - **Zoom controls** (+/- or a slider) to zoom/pan the view, unless pinch-to-zoom (for touch) and scroll (for desktop) suffice.
  - **Toggle relation types:** possibly a legend with checkboxes (e.g., [x] Article Links, [x] "See also" Links, [ ] Content Similarity) if such differentiation exists.
  - **Exit/Back:** A close button to return to the Reader view.
- **Interactions:**

- **Pan/Zoom:** Users can drag the canvas to pan around, and use scroll wheel or pinch to zoom in/out of the mesh to see more nodes or detail.
- **Hover/Tooltip:** On desktop, hovering over a node could display a tooltip with the article title (if not already labeled) and maybe a snippet of that article. On mobile, a tap could serve this, or tapping once focuses a node (and maybe shows a context menu).
- **Select/Navigate:** Clicking or double-clicking a node (or tapping on mobile) will navigate to that article's Reader view (closing the graph). Alternatively, a single tap could just highlight that node as the new focus and expand its neighbors (i.e., let user explore without leaving the mesh). We might support both: one gesture for "open article now" and another for "explore from this node". For instance, double-click opens article, single-click expands the graph around that node.
- Possibly **drag nodes:** to manually rearrange if the user wants to see a specific configuration (less important but can make the graph feel interactive).
- **States:**
- **Initial state:** When opened, graph is centered on current article node, with immediate links laid out around. If it takes time to compute, show a loading animation (maybe nodes appearing progressively).
- **Expanded state:** If user explores further, the graph might grow. We should keep performance in mind (limit number of nodes rendered to something manageable or paginate layers).
- **Empty state:** If an article has no links (unlikely in Wikipedia), show a message in the graph view ("No connected topics to display").
- **Error state:** If the semantic engine cannot fetch needed data (maybe certain analysis not available), we display an error or a message like "Semantic graph not available for this content."

## Bookmarks & History Views

- **Layout:** These can be simple list views. Possibly combined or separate.
- Bookmarks might be accessible via a "Bookmarks" section where all bookmarked articles are listed, grouped by archive if multiple.
- History could similarly list recently viewed pages, possibly grouped by date (Today, Yesterday, Last Week).
- **Key UI Elements (Bookmarks):**
  - List of bookmarked items. Each item shows the article title and the archive name (if multiple archives are in use), maybe an icon or favicon. Possibly also the date it was bookmarked if relevant.
  - Items are clickable to open the article. There could be an "edit" or "remove" action (a trash icon or swipe to delete on mobile).
  - If we allow folders/tags for bookmarks (maybe not initially), a UI to organize them.
- **Key UI Elements (History):**
  - Chronologically ordered list of articles viewed. Each entry shows title, archive, and timestamp (e.g., "Yesterday 14:30 - Article Title").
  - Possibly group by day for readability.
  - A "clear history" button at the top to wipe history.
- **Interactions:**
  - Click a bookmark or history entry -> open that article in Reader.
  - Swipe left (mobile) or a small "X" button to remove an entry from bookmarks or history.
  - Maybe long-press to get options like "Remove" or in bookmarks "Share" (to share the article link externally, if applicable).
- **States:**
- **Empty State:** If no bookmarks yet, show a placeholder ("No bookmarks yet – add some by tapping the star icon on any article!"). Similarly for history ("You haven't read any articles yet." if truly empty).

- **Loaded State:** Display the lists as described.
- These views are mostly static lists, so loading state not significant beyond maybe an initial load delay if data stored on disk (rare, since these are small sets).

## Layout and Styling Notes

- **Design System:** Use Tailwind CSS utility classes extensively to ensure consistent spacing, sizing, and colors. We will establish design tokens via CSS variables for theming (as done in the extension, which uses a 9-theme system with variables <sup>9</sup>). For example, define `--color-bg`, `--color-text`, `--color-accent` for theme colors and have Tailwind consume these via config for dynamic themes.
- **Grid/Flex Layouts:**
- Library view: likely a CSS grid with auto-fit columns for archive cards (e.g., `grid-cols-auto-fill minmax(200px, 1fr)` to make them responsive). On very small screens it becomes one column.
- Reader view: a flex column (top bar flex-none, content flex-auto scrollable). The content itself can be given padding and max-width for readability (like centered column on large screens).
- Search view: perhaps a flex column as well; or absolute positioned overlay. If overlay, use flex to center the no-results message, etc.
- Semantic mesh: fill the container; use absolute positioning for canvas; overlay controls positioned with flex or grid in a corner.
- Bookmarks/history: simple flex or block list with maybe dividers between days.
- **Component Reuse:** Define reusable components like `ZimCard` (for library entries), `ArticleListItem` (for search results, bookmarks, history entries), `TopBar`, `SearchBar` etc. This modular approach avoids duplicating style logic. Each should be designed in a responsive way (using Tailwind breakpoints).
- **Responsive Typography:** Ensure font sizes adjust for mobile vs desktop. Possibly use a slightly larger base font on mobile for readability. Tailwind can apply different text sizes at different breakpoints.
- **Icons and Visuals:** Use a consistent icon set (Lucide or similar, which the project already uses <sup>16</sup>). Icons needed: menu, back, forward, search, bookmark (filled/empty), graph, home/library, settings, etc. Keep icon style consistent (stroke width, size).
- **States Styling:**
- Buttons and interactive elements should have clear hover and active states (e.g., using Tailwind classes for `hover:bg-gray` etc. to show a highlight).
- For selected items (like the active archive or active link in some menu) use a distinct style (e.g., highlight or underline).
- For dark mode, ensure all text is readable (light text on dark background) and vice versa. Use CSS variables to swap theme colors without changing components.
- Loading indicators: could use a simple spinner component (Tailwind + an SVG or CSS animation). Possibly incorporate a skeleton screen for article content (grey bars representing text lines) if loading is noticeable.
- Error messages: style them consistently (e.g., red text or an alert box with an icon) in case of error states.

## Component States & Transitions

- **Loading State Example:** When switching between articles, especially if using a slow storage (like from an SD card on mobile), display a quick fading transition: the old content can fade out or overlay a translucent white, a spinner in the middle rotates until the new content fades in. This gives a smooth feel.

- **Semantic Node Hover:** On hovering a node in the semantic mesh (desktop), emphasize it and its direct edges. For example, the node enlarges slightly or changes color, and all connected edges are highlighted while others dim. A tooltip box appears near the node with the article title. On mobile, a tap could trigger this state (and maybe require a second tap to open the article).
  - **Active Tab/Section:** If using tabs (like Library, Bookmarks, Settings), the active tab should be visually distinct (highlighted icon or different background). Transitions between tabs can be animated (e.g., slide in for mobile, or a fade for desktop) to indicate a change of context.
  - **Error State in Components:** e.g., if a search query fails, the search view might show a message with a sad emoji and an explanation. If loading a ZIM file fails (corrupted file), the Library view might show an error badge on that archive.
  - **Empty States & Onboarding:** The very first run of the app might have no ZIMs. In that case, the Library view instead of an empty list could show a welcome message and instructions ("Get started by adding a ZIM archive"). Similarly, first time opening semantic mesh might show a hint overlay about what it is, which can be dismissed.
  - **Interactive States:** The UI should clearly reflect interactive states, e.g., when a bookmark is added, give quick feedback (toast notification "Bookmarked!" or briefly animate the icon). When a user toggles theme or font size in settings, apply it immediately to preview.
- 

## 4. Prompt/Agent Integration Layer

### Builder Agent Prompt Format

To streamline development with AI "builder agents", we establish a standard prompt format for requesting modules or components. All builder prompts will explicitly state the task, target component, inputs, and expected output context. The format will be:

**Builder Agent:** Construct [ComponentName] with [Inputs] and [Requirements]...

For example, to have an agent implement the library view component, the prompt might be:

**Builder Agent:** Construct `LibraryView` component with a grid of `ZimCard` elements as children, using input from an array of `ZimIndex` objects (each containing title, size, and id). Ensure responsive grid layout via Tailwind and include an "Add Archive" button at the top.

Each prompt starts with the key phrase *Builder Agent* to clearly indicate the role, followed by the specific instructions. The agent should output the requested code or structured result (UI markup, logic, etc.) according to the prompt.

### Naming Conventions for Modules & Components

Consistency in naming helps multiple agents work in concert:

- **Views and Pages:** Use `PascalCase` names ending with `View` for full pages/screens (e.g., `LibraryView`, `ReaderView`, `SearchView`, `MeshView`, `BookmarksView`). These typically correspond to a route or a major section of the app.
- **Reusable Components:** Use `PascalCase` without special suffix for smaller components (e.g., `ZimCard`, `SearchBar`, `TopNavBar`, `ArticleItem`). Their file names mirror the component name.
- **State and Models:** Use `camelCase` for variables and functions. For global state objects/types, use descriptive names (e.g., `AppState`, `ZimArchiveInfo`, `ArticleEntry`). Actions or events can be

named like `openArchive`, `navigateArticle`, `addBookmark` for clarity. - **Agent Task Identifiers:** When referring to tasks in documentation or commit messages, we can use tags like `[UI.LibraryView]` or `[State.Bookmarks]` to indicate the area. This helps agents (or developers) quickly locate relevant sections of this document to get context. For example, an agent working on bookmarks might look for the "BookmarksView" section. - **Folder/Directory Structure:** Organize the code so that agents know where to place files. For instance: - `frontend/src/views/LibraryView.tsx` for the main view components, - `frontend/src/components/ZimCard.tsx` for sub-components, - `frontend/src/state/store.ts` for the global state management. We communicate this structure in prompts when necessary (e.g., "Place the component in `components/ZimCard.tsx` and ensure it's exported").

By following these naming conventions, builder agents can generate code that fits seamlessly into the project, avoiding naming collisions or style inconsistencies.

## State Coordination Strategy

When multiple AI agents build different parts of the app, a unified state management approach ensures their outputs interoperate. We define a global application state (using a state container like Redux or Zustand, or even React Context for simplicity). Key aspects: - **Single Source of Truth:** There will be one global state object (or a few logically partitioned slices) that holds app-wide data: e.g., `currentZimId`, `openArticle`, `bookmarks[]`, `history[]`, `zimList[]`, and UI flags like `theme`. Agents building features will read/write via this state instead of each creating local state. - **State Access Patterns:** We'll create hooks or context providers for the state (e.g., `useAppState` hook) so that any component can access and modify global state in a standard way. Builder agents should use these provided hooks/actions. For example, an agent coding the `LibraryView` knows to dispatch an action `openArchive(id)` when a ZIM card is clicked, which sets `currentZimId` and loads the main page. - **Actions and Events:** Define a set of actions (for Redux) or functions (if using Zustand) that modify state. Examples: `loadZimList(files)`, `openArchive(id)`, `openArticle(articleId)`, `toggleBookmark(articleId)`, etc. Each agent implementing a feature will use these instead of creating new ones. We document these actions in an interface spec that all prompts reference, so an agent generating the `ReaderView` knows, for instance, that calling `toggleBookmark(currentArticle)` will update bookmarks. - **Cross-Module Communication:** For complex interactions (e.g., after adding a bookmark in `ReaderView`, the `BookmarksView` list should update), rely on state reactivity. Since both share the `bookmarks` state, the framework's reactive update will refresh the list. Agents thus focus on updating state correctly. - **Concurrency Considerations:** If multiple builder agents are used simultaneously (one might be working on the backend logic while another on UI), we ensure they have a locked interface contract. E.g., if one agent defines the state shape, others should not deviate. In practice, we might sequence their work or have a supervising agent/team lead to reconcile any differences. - **Documentation for Agents:** This document serves as a live spec. Each section (state shape, actions list) is clearly marked so an agent can be pointed here before generation. For example, a prompt might include: "Refer to the **State Coordination** section of the spec for available actions and state fields." This encourages consistency.

## Prompt Templates and Examples

To further guide builder agents, we provide stub templates for common tasks. These templates ensure the prompt covers all necessary details: - **UI Component Template:**

Builder Agent: Construct a `[ComponentName]` component.  
Description: [High-level description of component purpose].

Inputs: [Data inputs or props it receives].

Requirements:

- [UI/UX specifics: layout, styling cues, responsive behavior].
- [State interactions: what global state or actions it uses].
- [Any conditional rendering or sub-components].

Output: Provide the JSX/HTML and relevant logic for the component, using Tailwind CSS for styling.

*Example:* For Search Results list:

Builder Agent: Construct `SearchResultsList` component.

Description: Displays a list of article search results.

Inputs: `results` (Array of ArticleEntry objects) and `onSelect` (callback when an item is clicked).

Requirements:

- Each result item shows article title and snippet.
- Highlight query terms in bold.
- If no results, show a "No results" message.
- Use `
` list with Tailwind classes for spacing.

Output: JSX for the list and items, with event handling for selection.

- **Backend/Logic Module Template:** (if we have agents writing non-UI code, e.g., a search indexing function)

Builder Agent: Implement [FunctionName] in [File/Module].

Description: [What the function does, e.g., search or parse something].

Inputs: [Parameters].

Outputs: [Return value and its structure].

Constraints:

- [Performance or memory considerations].
- [Use of specific libraries or algorithms].

Provide code in the required language, with comments explaining key steps.

- **Integration Example:** We might have an agent integrate a UI with state:

Builder Agent: Integrate `BookmarksView` with global state.

Description: The BookmarksView should retrieve the list of bookmarks from global state and display them.

Steps:

1. Use the `useAppState` hook to get `bookmarks`.
2. If empty, show a placeholder.
3. Otherwise, render a list of links; clicking a link dispatches `openArticle(articleId)`.

Output: Modified `BookmarksView` component code with state integration.

- **Testing/Validation Template:** We also define how to prompt agents to write tests or verify functionality, but that might be beyond this PRD's scope. However, a similar pattern would be followed.

By following these templates, we ensure each agent receives clear, structured instructions, reducing ambiguity. All prompts will include references to this spec (section or line numbers if needed) to ground the agent in the project context.

## Example Stub Generation Prompt

To illustrate, here's a concrete prompt using our format:

**Prompt:**

```
Builder Agent: Construct `LibraryView`.  
Description: The main library screen showing all available ZIM archives in a responsive grid.  
Inputs: `zimList` (Array of ZimArchiveInfo objects with {id, title, size, description}).  
Requirements:

- Display a grid of ZimCards. On desktop, 3-column grid; on mobile, 1-column list.
- Each ZimCard shows title, description snippet, and size (in MB).
- Include an Add button at the top-right to import new archives.
- When a ZimCard is clicked, call `openArchive(id)` action and navigate to ReaderView.
- Use Tailwind for styling (card has border, padding, hover effect).



Output: Return a JSX component for LibraryView fulfilling above.


```

The agent would then produce the `LibraryView` component code, following our naming and state usage conventions. This approach can be repeated for each module (e.g., "Construct `MeshView`", "Construct `SearchBar`", etc.), enabling a collaborative build-out of the app.

---

## 5. Cross-Platform Considerations

### Desktop (Tauri) vs Web vs Mobile Differences

The project targets desktop, web, and mobile platforms, which necessitates some conditional handling:

- **Desktop (Tauri):** Runs as a native-like app with Rust backend. Benefits:
  - **File Access:** Can open ZIM files directly from disk (no size limit concerns). We implement file open via Tauri's dialog API <sup>26</sup> and keep the file path or handle in the Rust backend.
  - **Performance:** Heavy computations (indexing, large decompression) can be done in Rust threads, keeping the UI smooth.
  - **Integration:** Can use system features like a system tray for quick access to downloads/status <sup>27</sup>, and menu bar actions (standard File/Edit menu) <sup>28</sup>.
- **Distribution:** Packaged per OS, and users install it.
- **Web (Browser or PWA):** Runs in a sandboxed browser environment.
- **File Loading:** User must manually select a .zim file (via file input). The app then likely reads it into memory or partially uses IndexedDB. Large files (>1-2GB) might be challenging to handle purely in browser memory, so we might require the user to have enough RAM or use streaming (if the browser File API allows reading chunks incrementally without loading whole file at once).
- **No Native Code:** We rely on WebAssembly or JS for all logic. Our TS ZIM library is pure JS and should work, but performance might be lower than native. We ensure optimization in the library (e.g., using ArrayBuffer efficiently, perhaps web workers for search).
- **Offline Availability:** If we want a PWA, we must implement service workers to cache the app shell. The ZIM files themselves could be

stored in IndexedDB or kept as blobs across sessions. The Chrome extension variant uses Chrome storage (which might allow more persistent storage of the archive). - **Security:** Browsers will block certain things (like we can't write to arbitrary files, which is fine since we stay internal). Also, any web version might not allow saving a new ZIM file (except by offering it as a download to user). - **Mobile (Tauri Mobile / Hybrid):** - The Android version (using Tauri) will be similar to desktop in architecture: a Rust core and a WebView UI. However, mobile has constraints: e.g., need permission to access storage if we allow selecting external .zim files, handling different file paths (Android SAF). We might package some ZIM files with the app or require user to sideload. - **UI adjustments:** On mobile, we implement touch gestures (as noted) and likely use a simpler navigation (maybe a single-activity style where Library and Reader are in one stack). Tauri mobile is still evolving, but we anticipate using it to share codebase. - **Performance on mobile** might be more limited; we might avoid memory-intensive operations or offload them (perhaps use smaller ZIM slices or require using pre-indexed ZIMs on mobile). - **iOS:** If in future, possibly use a different container (Capacitor or React Native) if Tauri iOS isn't ready. Code reuse would still be possible by sharing the React components and maybe using the TS zim library via a WebView context.

## Code Reuse Strategies

To maximize cross-platform reuse: - **Single Codebase for UI:** The React/Tailwind UI is used for all targets. We do not maintain separate storyboards or native UI for mobile; instead, we ensure our web UI is responsive and touch-friendly. With Tauri enabling WebView UI on desktop and mobile, we essentially have a "write once" UI (with some responsive variants). - **Shared Logic:** The core logic (ZIM parsing, search, etc.) is implemented in portable libraries. We have it in TS (which runs on web) and in Rust (for Tauri). We might create a thin abstraction so that the React code calls an interface and under the hood it either calls a JS function or makes a Tauri RPC. For example, a function `loadArticle(zim, path)` could be provided by a module that checks if `window.__TAURI__` is present (signifying Tauri) and if so, calls the Rust command, otherwise uses the JS library. This way the React components do not worry about platform. - **Tailwind and CSS:** Using Tailwind ensures the styling is consistent across devices. We avoid platform-specific styling (no reliance on desktop-only hover effects for critical features – we ensure everything is also operable via click/tap). - **Conditional Features:** Some features might be desktop-only. For example, "Open in External Browser" or printing an article might not make sense on mobile or web. We will use conditional rendering based on platform. This can be as simple as checking user agent or environment variables (Tauri can provide a flag to the front-end). Builder agents coding features will be informed of such conditions (e.g., wrap a component in `{isDesktop && <DesktopOnlyFeature />}`). - **State Management Uniformity:** Using one state system (say Redux or Zustand) across all helps – the logic to add a bookmark is identical on each. Only the storage backend differs (desktop might write to a file or localStorage, web uses IndexedDB, etc., but that is abstracted behind the state actions). We can implement a persistence layer that uses appropriate API per platform yet exposes the same API to the rest of the app. - **Testing on all Platforms:** We will test components in a web browser for quick iteration (since it's the same React code), and also test the packaged desktop and mobile versions to catch any platform-specific issues (like file path handling or performance hiccups). This ensures our abstractions truly kept the platform differences minimal. - **Use of Web Standards:** For features like file download or share: - On web, use the HTML5 download attribute or navigator.share API for sharing content. - On desktop, use Tauri's APIs (e.g., invoke a save dialog or use OS share if available). - We encapsulate these in utility functions (e.g. a `saveFile(data, name)` function that uses `if (window.__TAURI__) ... else ...`). Agents writing such code will follow this pattern so the code stays DRY.

## Tailwind and Design Tokens

We leverage Tailwind's configuration to define a design token system that works everywhere:

- Define color themes in CSS (as mentioned) and use the `@apply` directive or custom classes if needed for thematic elements (e.g. a `.bg-base` class that maps to `bg-[var(--color-bg)]`). This ensures the theme switch logic is simple (update `--color-bg` etc. on the root).
- Use Tailwind's responsive variants extensively (`sm:`, `md:`, etc.) to adjust layout for mobile vs desktop.
- Use relative units or viewport units for some sizing so that on a small mobile screen things aren't too tiny or too large.
- On mobile, ensure any fixed element (like a bottom nav) uses safe-area insets (Tailwind has utilities or we can add CSS for `padding-bottom: env(safe-area-inset-bottom)` on iOS).
- Font selection: Possibly use system fonts for performance (like `-apple-system`, Roboto on Android, etc., which was noted in the API HTML snippet). This avoids needing custom font files and gives a native feel.

## State Store Implementation

We need a single state store accessible across components. Options:

- **Redux**: predictable and familiar, but requires boilerplate. Might be overkill for smaller app, but with many views and actions, it provides structure. Could be useful as multiple agents can easily follow a pattern (actions, reducers).
- **Zustand or Context**: simpler hook-based store. Zustand would allow direct usage in any component without drilling props.
- For now, assume we pick Zustand for simplicity (lightweight, no boilerplate). We'll have a `useStore` hook to get state and actions.
- **Persistence**: On app start, we load any saved state (like bookmarks) from local storage or a file into the store.
- We ensure that this state logic is included in the project so any agent can call e.g. `useStore.getState().bookmarks` or `useStore.getState().openArchive(id)`.
- Agents should avoid using browser-specific global storage directly; always go through this store interface. This way, switching the storage method (to a file on desktop, etc.) is centralized.

## Platform-Specific UI Adjustments

- As noted from the UI TODO <sup>8</sup>, mobile UI should have larger touch targets and possibly a different navigation layout (e.g., bottom nav). We will implement that by CSS and conditional rendering:
- Use Tailwind's `sm:` breakpoint to detect small vs large screens and show different nav arrangement (or use JS to detect if running on mobile device).
- For example, a sidebar on desktop might collapse into a bottom tab bar on phone.
- Some advanced desktop features (system tray, menu) are handled outside React via Tauri (Rust side creates a tray icon <sup>27</sup>, OS menu <sup>28</sup>). These don't exist on web, so we guard them. This doesn't affect the React UI much, but we note it in architecture (these features coded in Rust won't be invoked on web).
- On web, the extension's context menu actions (right-click to save page) exist, but in the app that concept is irrelevant. Those extension features remain only in the extension context (not in the standalone app), so we separate concerns. This PRD primarily covers the reader app; the extension is related but distinct. If we unify them, maybe the extension uses the same core library but a different UI surface (browser's popup and options pages).

In summary, we strive to write once and adapt via configuration. Platform differences are handled by capability detection and conditional code paths, all while using the same core logic and UI components across all targets.

---

## 6. Export/Import Support

### Export to Design Tools (Figma/Excalidraw)

To facilitate a design-development loop and allow visual refinement, the system includes hooks to export the current UI layouts or components to design tool formats:

- **Figma Export (HTML to Design):** We can integrate with tools like *html.to.design* <sup>29</sup> to convert our rendered React components into Figma frames. For example, a developer or designer could press an “Export to Figma” button which takes the current UI DOM, strips out interactive behavior, and uses the plugin/REST API to create a corresponding Figma document. This allows designers to fine-tune spacing, alignment, or propose changes in Figma based on the actual implementation. Because our UI is built with standardized web tech, using such an exporter yields a fairly accurate Figma representation of the app screens <sup>30</sup>.
- **Excalidraw Integration:** For quick collaborative diagramming (especially for things like the semantic mesh concept or flowcharts), we plan to support exporting certain structures to Excalidraw. For instance, the semantic graph of an article could be output as an Excalidraw JSON or SVG, which can be opened in the Excalidraw app for editing or presentation. Conversely, designers could sketch a wireframe in Excalidraw and we could partially import it. We have an “Excalidraw stub workspace” in the project to play with such ideas (ensuring diagrams can be versioned alongside code).
- **How Export Works:** Likely through a command or script. In a development mode, an agent or developer can run a command to snapshot a view. For Figma, using the Figma API requires authentication, so maybe we produce a `.json` or `.svg` that can be manually imported into Figma via the *html.to.design* plugin. For Excalidraw, since it’s open, we could directly generate a `.excalidraw` JSON file with elements corresponding to our layout boxes.
- **Use Cases:** Exported designs are useful for design review, generating marketing images, or onboarding new contributors with a visual map of the app. It also helps to ensure the AI builder agents’ output can be verified visually by designers.

### Import / Codegen from Design

- **HTML/CSS to Figma (Design) Import:** If a designer works on a new layout or style in Figma, we want to minimize hand-off friction. Tools like *design.to.code* or Figma’s code inspect can be leveraged. In our process, we might use AI to assist: for instance, feeding a Figma design (via its JSON export or API) to an agent to generate the corresponding JSX with Tailwind. While not fully automated in this PRD, we intend to keep the door open:
- We ensure our component structure in code is mirrored in design components (for example, a `ZimCard` component corresponds to a Figma component named “ZimCard” with analogous layers). This naming parity means a plugin or script can map Figma layers to our code components.
- If using builder agents for codegen, we could supply them with the design specs (like “the Figma frame has these elements... now produce code”). The consistent naming helps the agent identify where things go.
- **Layout Stubs for Generation:** In some cases, we might only have a high-level layout idea and want the agent to fill it in. We maintain *stub templates* for layouts – e.g., a textual representation of a screen’s hierarchy:

```
LibraryView:  
  Header: Title + AddButton  
  Grid:  
    ZimCard * N (for each archive)
```

These stubs can be given to a builder agent as a starting point. The agent then generates actual JSX code from that outline, applying the styles and state connections.

- **Codegen Workflow:** For a new feature, the workflow could be:
  - **Design:** Create or update a wireframe in Figma/Excalidraw for the feature.
  - **Export:** Use the export mechanism to get a representation (could be a PNG for reference or a structured data via `html.to.design`).
  - **Agent Generation:** Feed that into an AI builder agent with instructions to align with our coding standards. The agent produces a first draft of the component.
  - **Import Back:** Once code is running, if design changes are needed, export again to design to refine visuals.
  - **Automation vs Manual:** Not everything will be automated. The hooks we mention (to design tools) primarily empower human designers to work with AI developers in the loop. Over time, as builder agents get more context-aware, we might automate more (e.g., “Agent, here’s a Figma JSON for the new Settings screen, output the React code for it”).

## Maintaining Design-Dev Consistency

- We aim for a **single source of truth** for design tokens. For example, if a color or spacing is changed in Figma’s style guide, we update the Tailwind config or CSS vars accordingly, and vice versa. This consistency means design exports/imports won’t drift (the plugin would recognize the same color codes, etc.).
- The **collaborative canvas** nature of this document and workflow means that as features are specified here with IDs and structure, both the `builder agents` and `designers` can refer to the same section heading (e.g., an agent generating `SearchView` code has this spec and a designer can open Figma and ensure the frame “`SearchView`” matches).
- **Version Control for Design:** Any exported design artifact (like an Excalidraw JSON or a Figma file link) can be referenced in the repository (maybe stored in a `design/` folder or referenced by URL). This way, changes in design are tracked. Builder agents can even be instructed to look at the diff (if, say, a padding increased, the agent might update Tailwind classes from `p-4` to `p-6` accordingly).

In summary, the export/import support is about creating a tight feedback loop between code and design: - Use automation tools to capture UI into design files <sup>29</sup>. - Allow AI and human agents to generate code from high-level designs. - Ensure the UI structure in code reflects in design components for easy mapping. - This integrated approach will speed up UI refinement and maintain a high fidelity between intended design and actual implementation.

---

### Sources:

- Project README – Clean-room ZIM implementation overview <sup>1</sup>
  - ZIM file format & usage (Wikipedia, offline content) <sup>3</sup>
  - Project Architecture – ZIM spec, directory entries, and read flows <sup>21</sup> <sup>14</sup>
  - UI Development TODO – Feature list for viewer (search, bookmarks, history) <sup>2</sup>
  - UI Development TODO – Cross-platform (extension, desktop, Android) status <sup>31</sup>
  - UI Development TODO – Mobile UI adjustments (touch targets, bottom nav) <sup>8</sup>
  - Reddit discussion – Kiwix target users (offline access for Global South) <sup>10</sup>
  - Sigma.js example – Wikipedia article network (“semantic mesh” concept) <sup>4</sup>
  - `html.to.design` – HTML to Figma conversion tool description <sup>29</sup>
-

1 12 README.md

<https://github.com/rebots-online/AnZimmerman/blob/5465cf4816753cd9e7054a77617e7dfcbec9b0cd/README.md>

2 5 6 7 8 9 11 16 26 27 28 31 UI\_TODO.md

[https://github.com/rebots-online/AnZimmerman/blob/5465cf4816753cd9e7054a77617e7dfcbec9b0cd/TOOLS/UI\\_TODO.md](https://github.com/rebots-online/AnZimmerman/blob/5465cf4816753cd9e7054a77617e7dfcbec9b0cd/TOOLS/UI_TODO.md)

3 ZIM (file format) - Wikipedia

[https://en.wikipedia.org/wiki/ZIM\\_\(file\\_format\)](https://en.wikipedia.org/wiki/ZIM_(file_format))

4 A cartography of Wikipedia pages around data visualization - Sigma.js

<https://www.sigmaj.s.org/demo/>

10 Kiwix - Access Wikipedia (And More) With no Internet : r/selfhosted

[https://www.reddit.com/r/selfhosted/comments/r8bl81/kiwix\\_access\\_wikipedia\\_and\\_more\\_with\\_no\\_internet/](https://www.reddit.com/r/selfhosted/comments/r8bl81/kiwix_access_wikipedia_and_more_with_no_internet/)

13 20 zim\_api.go

[https://github.com/rebots-online/AnZimmerman/blob/5465cf4816753cd9e7054a77617e7dfcbec9b0cd/api/go/zim\\_api.go](https://github.com/rebots-online/AnZimmerman/blob/5465cf4816753cd9e7054a77617e7dfcbec9b0cd/api/go/zim_api.go)

14 15 17 18 19 21 22 23 24 25 ARCHITECTURE.md

<https://github.com/rebots-online/AnZimmerman/blob/5465cf4816753cd9e7054a77617e7dfcbec9b0cd/ARCHITECTURE.md>

29 Convert HTML to Design in Figma - Builder.io

<https://www.builder.io/blog/html-to-design>

30 html.to.design — Convert any website into fully editable Figma designs

<https://html.to.design/home/>