Codex Sinaiticus Reader — Spec & Ready JSON Package

Purpose: Create a high-quality, beautiful, offline-capable web + Android app that presents the surviving contents of Codex Sinaiticus and links to full public-domain Bible texts for missing verses/books. This document is a developer-facing specification and includes a recommended tech stack, features, data model, sample JSON, UX notes, legal/attribution checklist, and an execution roadmap.

---

1. Vision & Constraints

Vision: Make the Codex Sinaiticus accessible, searchable, and enjoyable to read. Present original Sinaiticus fragments faithfully, and provide optional full-text public-domain translations (KJV, Brenton LXX, ASV, WEB) for continuity when a passage is partial or missing. The app is strictly referential: it makes no doctrinal claims.

Key constraints: - All hosted text must be public domain or licensed for redistribution. - Codex Sinaiticus manuscript images: link to institutional viewers or obtain explicit permission before embedding. - App must work fully offline (local-hosted JSON + local search index). - Keep user privacy: no account, PII, or external trackers by default.

---

2. Product Requirements (MVP)

Essential features: - Browse by Book → Chapter → Verse - Display Sinaiticus text fragments exactly as preserved; clearly mark missing text with `[missing]` or `[…]` and show provenance metadata (folio, sigla). - Toggle: `Sinaiticus-only` / `Sinaiticus + Full Text (Brenton/KJV/ASV/WEB)` - Local full-text search with fuzzy search and filtering by testament, apocrypha, or early Christian writings - Offline hosting of all text; small footprint via compressed JSON/SQLite - Link or open external public-domain full texts (Project Gutenberg / archive.org) for user reference - Bookmarking, highlights, copy/share snippets (text-only) - Accessibility (screen reader semantics, adjustable font sizes, high-contrast themes)

Nice-to-have: - Side-by-side comparison view (Sinaiticus fragment vs chosen public-domain translation) - Manuscript image viewer (if permission obtained) or in-app webview to official Codex site - Per-verse metadata (manuscript certainty, lacunae notes, reconstructed text suggestions) - Light-weight offline analytic telemetry (opt-in only)

---

3. Recommended Tech Stack

General choices (prioritize speed-to-market & unified codebase): - **Language:** TypeScript - **UI framework (Web):** React with Next.js (static export / SSG) - **Mobile:** React Native with Expo (shared code + component library) - **State management:** Zustand or React Context for simple state; React Query (TanStack Query) for local caching patterns - **Storage (local hosting of texts):** - Bundle compressed JSON files in app assets for initial release. - For efficient search and local queries, use **SQLite** (via `expo-

`sqlite` or `react-native-sqlite-storage`) or **IndexedDB** on web with an IDB wrapper. - Also provide a small prebuilt **Lunr/Fuse.js** index for fast client search. - **Search:** Fuse.js for fuzzy search and Lunr for more advanced full-text indexing. Build index at build-time from JSON. - **UI Kit:** Tailwind CSS (web) + Tailwind-like utility in RN (Tailwind RN / Dripsy) or use a component library like **React Native Paper** + custom styles. - **Data pipeline / Content management:** Store canonical source texts in a Git repo as JSON. CI builds will generate optimized search indices and SQLite bundles. - **CI/CD:** GitHub Actions. Web deploy to Vercel or static host. Mobile builds via Expo Application Services (EAS) and Play Store / internal testing via Google Play Console (internal test track). - **Testing:** Jest + React Testing Library, Detox / Playwright for E2E. - **Analytics & Crash reporting:** Optional and opt-in only. Use Sentry (opt-in) or self-hosted Matomo.

Why this stack: Single-language (TS) across web & mobile speeds development. Expo simplifies Android builds. Next.js gives a great web dev DX and ability to provide a static site for discovery.

---

4. Data Model (JSON schema)

Goals: Represent partial verses, lacunae, and alternate full-text links. Maintain small size and efficient parsing.

Top-level package layout (in repo / bundled in app):

```
/codex-package/
  metadata.json
  books/ (one file per book, e.g. matthew.json)
  indices/ (lunr index, fuse index)
  license/ (source attributions, TXT files)
```

Book JSON schema (example):

```
{
  "id": "john",
  "name": "John",
  "order": 43,
  "section": "New Testament",
  "folios": [{"folio": "f146r", "note": "Gospel of John start"}],
  "chapters": [
    {
      "chapter": 1,
      "verses": [
        {
          "verse": 1,
          "sinaiticus": {
            "text": "Ἐν ἀρχῇ ἦν ὁ λόγος.",
            "fragment": false,
            "notes": null
          },
          "fulltext_links": [
            {"lang":"en","translation":"KJV","url":"/full/kjv/john/1/1"}
          ],
          "packed_text": "In the beginning was the Word.",
          "is_partial": false
        },
        {
          "verse": 7,
          "sinaiticus": {
            "text": "... [fragment: ἐγένετο ἄνθρωπος ἀπεσταλμένος παρὰ
θεοῦ]",
            "fragment": true,
            "notes": "leaf damaged: only part preserved"
          },
          "fulltext_links": [ {"lang":"en","translation":"KJV","url":"/full/
kjv/john/1/7"} ],
          "packed_text": null,
          "is_partial": true
        }
      ]
    }
  ]
}
```

Notes on fields: - `sinaiticus.text` should store the original Greek fragment (or a transliteration) and must not be modified. If you offer an English translation of the fragment, save under `packed_text` with clear provenance. - `is_partial` indicates gaps where full verse text is not preserved. - `fulltext_links` point to local packaged public-domain translations (preferred) or externally to Project Gutenberg/archive.org if you choose linking instead of hosting.

---

5. Sample JSON snippet (public-domain full text mapping)

Below is a tiny sample of how the packaged public-domain KJV/Brenton entries might look (book→chapter→verse) for embedding locally. In the real package every book will be a separate file.

```
{
  "id": "psalms",
  "name": "Psalms",
  "chapters": [
    { "chapter": 1, "verses": [ {"verse":1, "text":"Blessed is the
man..."} ] }
  ],
  "source": {"translation":"Brenton LXX","year":1851, "license":"Public
Domain"}
}
```

---

6. UX / UI Patterns

- Clean reading view: large type, margin notes collapsed by default, swipe to change chapter.
- Fragment indicators: inline marker `[…]` with tooltip/modal showing manuscript folio and transcription.
- Toggle button in header: `Show full text` (when ON, app fetches local fulltext for missing/ partial verses and displays them faded or in a different color so users can see what is Sinaiticus vs added text).
- Search: quick fuzzy search across both Sinaiticus fragments and fulltext. Show search snippets with highlight and a badge `Sinaiticus` or `Full text (KJV)`.

---

7. Legal & Attribution Checklist (must ship with app)

- Include `LICENSES.md` listing every translation used, source URL, edition, and proof of PD status (Project Gutenberg / archive.org links).
- `ABOUT` screen: explain project goals, "no doctrinal claims", and provide contact info for takedown or corrections.
- If using CodexSinaiticus.org images or data, include copy of permission or a link to their copyright page and follow their terms (non-commercial restrictions if any).
- Add credits to Brenton (1851), Anderson (1918), KJV/ASV as applicable.

---

8. Packaging & Builds

Web: Build-time pipeline generates static JSON + Lunr index → Next.js static pages for SEO (book-level landing pages) → static export and host on Vercel or S3/Netlify.

Mobile (Expo): Bundle JSON assets with app (or download on first-run into local SQLite); at build time produce a pre-seeded SQLite DB that the app opens read-only.

Search index: Precompute Lunr/Fuse index at build time and bundle as compressed asset. On first-run, expand into IndexedDB/SQLite.

---

9. Developer Tasks & Deliverables

MVP deliverables: - Canonical codex JSON package (all Sinaiticus books/chapters/verses with `is_partial` flags). - Public-domain full-text JSON package (Brenton/KJV/ASV/WEB as chosen) with consistent verse keys. - Next.js web reader app (static export) and React Native + Expo Android app with reading & search features. - LICENSES.md and ABOUT page + in-app attributions.

Recommended next steps: - Choose primary public-domain translation(s) to host (Brenton recommended for LXX/apocrypha; KJV/ASV/WEB for general users). - Prepare canonical JSON for Sinaiticus: extract the per-verse fragment data and mark lacunae. - Build indexing pipeline (script to produce Lunr + SQLite DB). - Implement reader UI + search.

---

10. Final Notes & Open Questions

- Decide whether to **host full texts locally** (better UX) or **link externally** (simpler legal posture but dependent on remote sites).
- Plan for future features: side-by-side critical apparatus, commentary layers (licensed), community annotations (moderated), and scholarly exports (TEI/XML).

I included sample JSON examples and a full data model in this document. If you'd like, I can now generate the full starter JSON package (one book or small set) you can drop into the app repo — specify which books/translations you want first and I will output the JSON files.