Read-Later Engine — Full Project Canvas

Brief explanation (one line): This canvas is a single, complete engineering specification and implementation guide for the Read-Later Engine built with Next.js (App Router), NeonDB (Postgres), NextAuth (Google/GitHub/Email), a PWA (share target), and a small Firefox extension.

Table of contents

---

1. Overview, Motive & Goals

Project motive

The Read-Later Engine solves a common cognitive problem: users save articles for "later" and then forget them. The motive is to build a compact, personal productivity tool that captures those URLs across devices and nudges users to actually read them. For you, this is both a practical personal app and a foundational module for a larger "Life OS" (tasks, meetings, timelines) — so the project serves two purposes:

1. **Immediate utility:** quick wins — save links from the browser or share-sheet and get reminded.
2. **Platform foundation:** a reusable capture + reminder engine you can extend into task/meeting timeline features later.

Goals (refined)

- Fast capture: one-click save from desktop or mobile.
- Secure personal storage: only authenticated users can save/read items.
- Lightweight metadata: title, summary, reading time, tags.
- Cross-device sync and installable PWA UX.
- Extensible data model to add tasks/meetings later.

Feasibility analysis

Technical feasibility

- **Frontend / Backend:** Next.js (App Router) supports both UI and server routes; it's well-suited for this single-codebase approach. Feasible.
- **Database / Scaling:** NeonDB is serverless Postgres; it works with Next.js serverless patterns and scales for the expected load (personal to small team). Feasible.
- **Auth:** NextAuth integrates with Next.js and supports OAuth + email; implementation is straightforward. Feasible.
- **Browser integration:** Firefox desktop extensions are standard; Android can use PWA share targets. Feasible.
- **PWA features:** Installable, share target, and basic offline shell are supported by modern browsers and Vercel hosting. Feasible.

Operational feasibility

- **Development effort:** Single developer MVP (auth + save flow + inbox + PWA) estimated at 6–8 days for a working prototype; full polish with extension + reminders + deploy ~2–3 weeks. Reasonable.
- **Maintenance:** Low maintenance for a single-user app; adding multi-user or heavy scraping increases costs and monitoring needs.

Risks & Mitigations

- **SSR Fallback / Scraping issues:** Some sites block automated metadata fetchers. Mitigation: fall back to fetching title from HTML or let the user edit the title.
- **Session/cookie edge cases in extensions:** Cookies and same-site policies can interfere with extension auth. Mitigation: test auth flows, and provide explicit OAuth popup inside extension if needed.
- **Chrome mobile extension support lacking:** Mitigation: PWA share-target replaces mobile extension use-case.
- **Cost:** Neon/Vercel free tiers are generous for personal use; scale costs as user base grows.

---

2. High-level architecture

```
User Device (Browser/PWA/Extension)
  ↕ (HTTPS + cookies/session)
```

```
Next.js App (Vercel)
  - App Router pages
  - API Route Handlers (/api/*)
  - Auth (NextAuth)
  - Share receiver (/share)
  ↕
NeonDB (Postgres)
  - NextAuth tables
  - saved_items table
Background:
  - Vercel cron (reminders)
  - Optional: worker for heavy metadata extraction
```

Key flows: - Save via Firefox extension -> POST /api/items (authenticated) - Save via mobile -> Share -> PWA -> POST /share -> redirects -> POST /api/items - Read/Archive via UI -> PATCH /api/items/:id

---

3. Technology stack

- **Framework:** Next.js 14 (App Router)
- **Language:** TypeScript
- **UI:** React + Tailwind CSS
- **Auth:** NextAuth (Auth.js) with providers: GitHub, Google, Email Magic Link
- **DB:** NeonDB (Postgres)
- **ORM:** Drizzle ORM (recommended) or Prisma (optional)
- **Hosting:** Vercel
- **PWA:** next-pwa or manual service worker
- **Metadata:** open-graph-scraper (OGS) or node-fetch + parse
- **Extension:** Firefox Manifest V3 (simple popup)
- **Background jobs:** Vercel Cron (or an external worker / serverless function)

---

4. Authentication design

Providers

- **Google OAuth** — for general users and convenience.
- **GitHub OAuth** — for developer-friendly login.
- **Email Magic Links** — passwordless backup login.

Why NextAuth

NextAuth (Auth.js) integrates with Next.js App Router, has adapters for Postgres (works with Prisma/Drizzle), and simplifies session handling and provider setup.

Session & API protection

- Use `middleware.ts` to protect `/api/:path*` using `next-auth/middleware`.
- Server-side route handlers should validate session using `getServerSession()` before performing DB operations.

Environment variables (minimum):

```
DATABASE_URL=postgres://... (Neon)
NEXTAUTH_SECRET=... (random long string)
GITHUB_CLIENT_ID=
GITHUB_CLIENT_SECRET=
GOOGLE_CLIENT_ID=
GOOGLE_CLIENT_SECRET=
EMAIL_SERVER=
EMAIL_FROM=
NEXTAUTH_URL=https://yourapp.vercel.app
```

---

5. Database schema (Drizzle / Prisma-ready)

This is a clear, minimal schema. Keep NextAuth tables as provided by adapter.

saved_items table

- id: UUID PK
- user_id: UUID FK -> users.id
- url: TEXT
- title: TEXT (nullable)
- summary: TEXT (nullable)
- domain: TEXT
- favicon_url: TEXT (nullable)
- image_url: TEXT (nullable)
- reading_time: INT (minutes) (nullable)
- tags: TEXT[] (nullable)
- status: TEXT ("unread" | "read" | "archived") default: "unread"
- created_at: TIMESTAMP default now()
- remind_at: TIMESTAMP nullable

Drizzle example

```
import { pgTable, text, uuid, timestamp, integer } from "drizzle-orm/pg-core";

export const savedItems = pgTable("saved_items", {
  id: uuid("id").defaultRandom().primaryKey(),
  userId: uuid("user_id").notNull(),
  url: text("url").notNull(),
  title: text("title"),
  summary: text("summary"),
  domain: text("domain"),
  faviconUrl: text("favicon_url"),
  imageUrl: text("image_url"),
  readingTime: integer("reading_time"),
  tags: text("tags").array(),
  status: text("status").default("unread"),
  createdAt: timestamp("created_at").defaultNow(),
  remindAt: timestamp("remind_at"),
});
```

---

6. API contract (endpoints & examples)

Base URL: `https://yourapp.vercel.app/api`

POST /api/items

Purpose: Save a URL for logged-in user. **Auth**: Required (session cookie). **Request body**:

```
{ "url": "https://example.com/post", "tags": ["react","arch"] }
```

Response: `201` with saved item JSON.

GET /api/items

Query params: `status`, `search`, `tags`, `limit`, `offset` **Response**: list of items (paginated)

PATCH /api/items/:id/read

Purpose: mark as read **Body**: `{}`

PATCH /api/items/:id/archive

Purpose: archive

PATCH /api/items/:id/snooze

Body: `{ "remind_at": "2025-01-01T08:00:00Z" }`

DELETE /api/items/:id

Purpose: delete the saved item

POST /api/metadata (internal, optional)

Purpose: fetch metadata server-side for a URL (OG tags, reading time) **Body**: `{ "url": "https://..." }`

---

7. Next.js project structure (recommended)

```
/read-later-app
  /app
    /api
      /auth
        [...nextauth]/route.ts
      /items/route.ts
      /share/route.ts
    /inbox/page.tsx
    /login/page.tsx
    /settings/page.tsx
    /_middleware.ts
  /components
    Header.tsx
    ItemCard.tsx
    SaveButton.tsx
  /db
    index.ts
    schema.ts
  /lib
    metadata.ts
    auth.ts
  /public
    /icons
    manifest.webmanifest
    sw.js
  next.config.js
  package.json
  tailwind.config.js
```

Notes: - Use modular UI components. - Keep server-only helpers under `/lib` or `/server` and import only in route handlers.

---

8. Step-by-step project setup (commands + file seeds)

This section contains terminal commands and minimal file contents to get a working prototype.

8.1 Create project

```
npx create-next-app@latest read-later --ts
cd read-later
# choose App Router, TypeScript, Tailwind, ESLint
```

8.2 Install packages

```
npm install next-auth @neondatabase/serverless drizzle-orm drizzle-kit open-graph-scraper node-fetch cookie
# if using next-pwa
npm install next-pwa
```

8.3 .env.local (local dev)

```
DATABASE_URL=postgres://<neon-connection-string>
NEXTAUTH_SECRET=<random-string>
NEXTAUTH_URL=http://localhost:3000
GITHUB_CLIENT_ID=
GITHUB_CLIENT_SECRET=
GOOGLE_CLIENT_ID=
GOOGLE_CLIENT_SECRET=
EMAIL_SERVER=
EMAIL_FROM=
```

8.4 Drizzle DB connection (`/db/index.ts`)

```
import { drizzle } from "drizzle-orm/neon-http";
import { neon } from "@neondatabase/serverless";
const sql = neon(process.env.DATABASE_URL!);
export const db = drizzle(sql);
```

8.5 Create schema (`/db/schema.ts`) — see section 5

8.6 Migrations (drizzle-kit)

```
npx drizzle-kit generate
npx drizzle-kit push
```

8.7 NextAuth route (`/app/api/auth/[...nextauth]/route.ts`)

```
import NextAuth from "next-auth";
import GitHub from "next-auth/providers/github";
import Google from "next-auth/providers/google";
import Email from "next-auth/providers/email";
import { DrizzleAdapter } from "@auth/drizzle-adapter"; // hypothetical; adapt
to chosen ORM

export const authOptions = {
  providers: [
    GitHub({ clientId: process.env.GITHUB_CLIENT_ID!, clientSecret:
process.env.GITHUB_CLIENT_SECRET! }),
    Google({ clientId: process.env.GOOGLE_CLIENT_ID!, clientSecret:
process.env.GOOGLE_CLIENT_SECRET! }),
    Email({ server: process.env.EMAIL_SERVER, from: process.env.EMAIL_FROM }),
  ],
  secret: process.env.NEXTAUTH_SECRET,
  // adapter: DrizzleAdapter(db), // configure appropriate adapter for Drizzle/
Prisma
};

const handler = NextAuth(authOptions);
export { handler as GET, handler as POST };
```

8.8 Protect API with middleware (`middleware.ts`)

```
export { auth as middleware } from "next-auth/middleware";
export const config = { matcher: ["/api/:path*"] };
```

8.9 Basic save route (`/app/api/items/route.ts`)

```
import { getServerSession } from "next-auth/next";
import { db } from "@/db";
import { savedItems } from "@/db/schema";

export async function POST(req: Request) {
  const session = await getServerSession();
  if (!session) return new Response("Unauthorized", { status: 401 });

  const { url, tags } = await req.json();
  // Basic insert; metadata extraction deferred to separate step
  const res = await db.insert(savedItems).values({ userId: session.user.id,
url, tags });
  return new Response(JSON.stringify({ ok: true }), { status: 201 });
}
```

8.10 Inbox page (`/app/inbox/page.tsx`) — server component

```
import { db } from '@/db';
import { savedItems } from '@/db/schema';

export default async function Inbox() {
  // server-side fetch
  const items = await db.select().from(savedItems).limit(50);
  return (
    <div className="p-4">
      <h1 className="text-2xl">Inbox</h1>
      <ul>{items.map((it) => <li key={it.id}>{it.title ?? it.url}</li>)}</ul>
    </div>
  );
}
```

---

9. Metadata extraction (OpenGraph + fallback)

Strategy

- Use `open-graph-scraper` (OGS) server-side to fetch OG title/description/image.
- If OG missing, fetch HTML and extract `<title>` and first paragraph or compute summary from body text.
- Estimate reading time: `reading_time = Math.max(1, Math.ceil(wordCount / 200))` (200 wpm baseline).

Example implementation (`/lib/metadata.ts`)

```
import ogs from 'open-graph-scraper';

export async function fetchMetadata(url: string) {
  try {
    const { result } = await ogs({ url, timeout: 10_000 });
    const title = result.ogTitle || result.twitterTitle || result.title;
    const description = result.ogDescription || result.description;
    const image = result.ogImage?.url || result.twitterImage?.url;
    // fallback: fetch HTML and parse content for summary/word count
    const readingTime = estimateReadingTime(description || "");
    return { title, description, image, readingTime };
  } catch (err) {
    return { title: null, description: null, image: null, readingTime: null };
  }
}

function estimateReadingTime(text: string) {
  const words = text.split(/\s+/).filter(Boolean).length;
  return Math.max(1, Math.ceil(words / 200));
}
```

Note: For heavy scraping or unreliable sites consider using a queue + worker to avoid blocking API responses.

---

10. PWA manifest & share-target

Place `manifest.webmanifest` under `/public`.

```
{
  "name": "Read Later",
  "short_name": "ReadLater",
  "start_url": "/inbox",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#0b1220",
  "icons": [
    { "src": "/icons/icon-192.png", "sizes": "192x192", "type": "image/png" },
    { "src": "/icons/icon-512.png", "sizes": "512x512", "type": "image/png" }
  ],
  "share_target": {
    "action": "/share",
    "method": "POST",
    "enctype": "multipart/form-data",
    "params": {
      "title": "title",
      "text": "text",
      "url": "url"
    }
  }
}
```

Share target handler (`/app/share/route.ts`)

- Accept `POST` multipart/form-data
- Extract `url`, `title`, `text`
- If user has session cookie (PWA installed and logged in), create saved item
- If no session, redirect to login then back to share flow

Example minimal route (pseudo):

```
export async function POST(req: Request) {
  // parse form-data
  // create saved item or redirect
}
```

---

11. Service worker & caching strategy

Goals

- Make the app installable and provide an offline shell (view previously loaded inbox items).
- Support basic static asset caching and runtime caching for API calls (optional: cache read items for offline view).

Approach (next-pwa or manual)

- Precache: HTML shell, CSS, JS bundles, core icons
- Runtime cache: API responses for `/api/items` with short TTL
- Background sync: (future) store outgoing saves when offline and sync on reconnect

Note: Implement background sync carefully — service worker storage + IndexedDB will be required.

---

12. Firefox extension (manifest, popup, background)

A minimal extension: a popup with one button to "Save". It reads the active tab URL and posts to your API.

manifest.json (MV3)

```json
{
  "manifest_version": 3,
  "name": "Read Later Saver",
  "version": "1.0",
  "action": { "default_popup": "popup.html" },
  "permissions": ["tabs", "cookies", "storage"],
  "host_permissions": ["https://yourapp.vercel.app/*"]
}
```

popup.html

```html
<!doctype html>
<html>
  <body>
    <button id="save">Save To ReadLater</button>
    <script src="popup.js"></script>
  </body>
</html>
```

popup.js

```javascript
async function save() {
  const tabs = await browser.tabs.query({ active: true, currentWindow: true });
  const tab = tabs[0];
  await fetch('https://yourapp.vercel.app/api/items', {
    method: 'POST',
    credentials: 'include',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({ url: tab.url })
  });
  alert('Saved');
}

document.getElementById('save').addEventListener('click', save);
```

Important: For the extension to be authenticated automatically it must share cookies with the site domain (default behavior). If using cookie restrictions, you may need to use OAuth popup flow inside the extension.

---

13. Security considerations

- Use HTTPS everywhere (Vercel handles this)
- Keep `NEXTAUTH_SECRET` strong and private
- Use short-lived sessions and rotate refresh tokens (NextAuth handles this)
- Validate and sanitize URLs before fetching metadata (avoid SSRF)
- Rate-limit metadata endpoint to avoid being used as an open proxy
- When parsing HTML, avoid executing scripts; treat it as plain text
- CORS: restrict API origin if exposing endpoints beyond your domain

---

14. Deployment (Vercel + Neon) checklist

1. Create NeonDB project and database
2. Create Vercel project and connect to GitHub repo
3. Add environment variables in Vercel dashboard (see section 4)
4. Configure NextAuth providers (set OAuth callback URLs in GitHub/Google consoles to `https://yourapp.vercel.app/api/auth/callback/...`)
5. Configure PWA assets (icons) and publish
6. Publish Firefox extension (optional) — follow Mozilla docs for listing
7. Test share target on Android (install PWA, share from Chrome)
8. Run drizzle migrations against Neon in production

---

15. Developer workflow & testing

- Local dev: `npm run dev` (Next.js) and use `ngrok` for testing share flows if needed
- DB migrations: `npx drizzle-kit generate` / `npx drizzle-kit push`
- Test authentication: use `localhost` redirect URIs in OAuth app configs during dev
- Use Postman / HTTPie to test protected API routes (copy session cookie from browser)
- Use browser extension `about:debugging` in Firefox to load unpacked extension for testing

---

16. Milestones & timeline (suggested minimal schedule)

- **Day 1:** Project bootstrap, NeonDB connection, basic schema, NextAuth email magic link
- **Day 2:** POST /api/items, basic metadata extraction, Inbox UI (server component)

- **Day 3:** PWA manifest + share target, test Android share flow
- **Day 4:** Firefox extension (popup) and extension auth test
- **Day 5:** Add GitHub + Google OAuth providers, polish UI, basic tests
- **Day 6:** Add Drizzle migrations, deploy to Vercel, test production flows

Adjust timeline to your available time — this is a one-developer MVP schedule.

---

17. Future features & roadmap

- Smart reminders + schedule engine
- AI summarization of articles
- Auto-tagging via embeddings
- Cross-device read progress + highlights
- Calendar sync for meetings & timelines (extend the scheduler service)
- Public/private collections + sharing between users
- Analytics & habit tracking (streaks, weekly digest)

---

18. Appendix: useful snippets and templates

next.config.js (PWA + App Router)

```
const withPWA = require('next-pwa')({ dest: 'public', register: true,
skipWaiting: true });
module.exports = withPWA({ experimental: { appDir: true } });
```

Example POST /api/items (full with metadata)

```
import { getServerSession } from 'next-auth/next';
import { db } from '@/db';
import { savedItems } from '@/db/schema';
import { fetchMetadata } from '@/lib/metadata';

export async function POST(req: Request) {
  const session = await getServerSession();
  if (!session) return new Response('Unauthorized', { status: 401 });

  const { url, tags } = await req.json();
  const meta = await fetchMetadata(url);
  const inserted = await db.insert(savedItems).values({
    userId: session.user.id,
    url,
    title: meta.title,
    summary: meta.description,
    imageUrl: meta.image,
    readingTime: meta.readingTime,
    tags,
  }).returning();

  return new Response(JSON.stringify(inserted), { status: 201 });
}
```

Final notes

This canvas is intended to be the single source of truth while you build the Read-Later Engine. Use it to scaffold your repository, onboard collaborators, or generate project tasks.

When you are ready I can: - Generate the full repo folder structure as file templates in the canvas, - Create the Firefox extension files and a ZIP you can test, - Produce the exact NextAuth config (code block) with the adapter for Drizzle or Prisma, - Generate a minimal inbox UI with components and Tailwind styles, - Produce the Vercel environment variable list and a CI/CD checklist.

Pick one of the above and I will add it directly into this canvas as another page (or expand this document).