**ToolDeck — Project Architecture & File Structure**

This document describes a **developer‑friendly file and folder architecture** for the offline, cross‑platform Electron PDF reader (Firefox‑style) you want to build. It focuses on maintainability, security, testability, and incremental development.

**Principles**

* **Offline first:** bundle all runtime assets (PDF.js, workers, libs) locally. No CDN. No network calls by default.
* **Privacy by design:** no telemetry, no analytics, no data exfiltration. Open source under MPL.
* **Secure Electron setup:** contextIsolation: true, nodeIntegration: false in renderers, a small, audited preload.js bridge exposing only required APIs.
* **Modular & incremental:** separate PDF rendering (PDF.js) from annotation logic and file IO. Let renderer hold UI and viewer logic; main process handles native dialogs and process management.
* **Pure HTML/CSS/JS:** no frameworks, keep modules small and testable.

**Top-level project tree (recommended)**

/tooldeck

├─ .github/ # CI workflows, issue/PR templates

├─ assets/ # icons, fonts, sample PDFs used for development

├─ build/ # build output (packaged apps)

├─ docs/ # design docs, wireframes, decision logs

├─ scripts/ # helper scripts (port finder, spawn helpers)

├─ src/

│ ├─ main/ # Electron main process code

│ │ ├─ main.js # entry point used by package.json (or index.js)

│ │ ├─ menus.js # app menus & accelerators

│ │ ├─ ipc-handlers.js # main-side IPC handlers

│ │ └─ backend-manager.js# spawn/kill ASP.NET core backend (future)

│ ├─ preload/ # secure bridge exposed to renderer

│ │ └─ preload.js

│ ├─ renderer/ # UI (pure HTML/CSS/JS) and app logic

│ │ ├─ index.html

│ │ ├─ styles/ # css, variables, icons sprite

│ │ │ └─ main.css

│ │ ├─ app.js # entry for renderer; tab manager + global state

│ │ ├─ tabs/ # tab manager + tab UI

│ │ │ ├─ tabManager.js

│ │ │ └─ tab.html

│ │ ├─ toolbar/ # toolbar implementation & actions

│ │ ├─ sidebars/ # thumbnails, outline, attachments

│ │ ├─ viewers/ # viewer wrapper modules (PDF.js encapsulation)

│ │ │ └─ PdfViewer.js # per-tab viewer class

│ │ ├─ tools/ # annotations & editing tools

│ │ │ ├─ highlight.js

│ │ │ ├─ ink.js

│ │ │ ├─ textNote.js

│ │ │ └─ signature.js

│ │ ├─ services/ # local services (serialization, persistence)

│ │ │ ├─ fileService.js # uses preload bridge to read/write disk

│ │ │ └─ settings.js

│ │ └─ utils/ # shared helpers (math, DOM helpers)

│ ├─ pdf/ # bundled PDF.js runtime & worker

│ │ ├─ pdf.js

│ │ ├─ pdf.worker.js

│ │ └─ pdf.worker.js.map

│ └─ libs/ # third-party libs bundled locally

│ ├─ pdf-lib.min.js

│ └─ localforage.js

├─ test/ # unit & e2e tests

│ ├─ unit/

│ └─ e2e/

├─ .eslintrc.json

├─ .prettierrc

├─ package.json

├─ README.md

└─ LICENSE (MPL)

Note: you can keep main.js at project root if you prefer the simpler layout. Putting it under src/main is cleaner for larger projects — if you move it, update package.json "main" field accordingly.

**Key files explained**

* **src/main/main.js** — Creates BrowserWindow(s), sets webPreferences (preload path, contextIsolation), registers app lifecycle events, and coordinates starting/stopping the optional backend.
* **src/preload/preload.js** — The only script that receives Node in renderer context. Exposes a minimal, strongly typed API via contextBridge.exposeInMainWorld('electronAPI', { ... }). Example methods: openFile(), saveFile(), showOpenDialog(), showSaveDialog(), readFile(path), writeFile(path, bytes), getFreePort(), spawnBackend(cmd, args), killBackend(pid), onBackendStatus(cb).
* **src/renderer/app.js** — App bootstrapping, initializes TabManager, sets up global keyboard shortcuts, loads user settings, and mounts UI.
* **src/renderer/viewers/PdfViewer.js** — Encapsulates the PDF.js viewer instance. Provides a clean API:
  + loadDocument(source) — accepts ArrayBuffer or file:// path
  + renderPage(pageNum)
  + getPageCount()
  + findText(query, options)
  + setZoom(level) / fitToWidth() / fitToPage()
  + rotate(deg)
  + exportPageAsImage(pageNum)
  + getTextContent(pageNum)
  + getAnnotations() / applyAnnotations(annotations)
  + destroy()
* **src/renderer/tools/** — Each tool is a module that registers itself with the active viewer instance. They should be small, stateless where possible, and emit annotation JSON back to the annotation service.
* **src/renderer/services/fileService.js** — Wraps electronAPI calls to read/write files and manage save dialogs. Also provides helper to export PDF bytes after flattening annotations.
* **src/pdf/** — Locally bundled PDF.js files. Configure the worker path at runtime, e.g.: pdfjsLib.GlobalWorkerOptions.workerSrc = 'src/pdf/pdf.worker.js' (resolve appropriately in production builds).

**Annotation model (suggested JSON schema)**

Store annotations as JSON in a simple portable format. This allows session restore, undo/redo, and exporting/importing annotations separately from PDFs.

{

"annotations": [

{

"id": "uuid-v4",

"type": "highlight",

"page": 3,

"rect": [x, y, width, height],

"quads": [ /\* optional: detailed selection quads \*/],

"color": "#FFEB3B",

"opacity": 0.8,

"author": "User",

"createdAt": "2025-09-01T12:00:00Z",

"modifiedAt": "2025-09-01T12:00:00Z",

"meta": { "custom": true }

}

]

}

Types to support initially: highlight, underline, strikeout, ink (freehand path), freetext, image, signature, stamp.

When saving into a final PDF, you have two options:

1. **Embed (flatten)**: use pdf-lib to draw annotation visuals onto each page's content, creating a new PDF file with annotations rasterized/drawn into page content.
2. **Save as separate annotations file**: keep original PDF untouched and write filename.pdf.annotations.json alongside it. This is useful for reversible edits.

For Firefox parity, implement both: by default keep annotations editable (JSON sidecar); provide a "Save as flattened PDF" option.

**IPC / Preload API design (recommended minimal surface)**

Expose a small, predictable set of APIs from preload to renderer:

contextBridge.exposeInMainWorld('electronAPI', {

openFile: () => ipcRenderer.invoke('dialog:openFile'),

saveFile: (defaultPath) => ipcRenderer.invoke('dialog:saveFile', defaultPath),

readFileBytes: (path) => ipcRenderer.invoke('file:readBytes', path),

writeFileBytes: (path, bytes) => ipcRenderer.invoke('file:writeBytes', path, bytes),

getFreePort: () => ipcRenderer.invoke('system:getFreePort'),

spawnBackend: (cmd, args, opts) => ipcRenderer.invoke('backend:spawn', cmd, args, opts),

killBackend: (pid) => ipcRenderer.invoke('backend:kill', pid),

on: (channel, cb) => ipcRenderer.on(channel, cb)

})

**Security notes:** validate inputs on the main process side. Do not expose require in renderer. Keep the bridge methods promise-based (invoke/handle) so errors propagate cleanly.

**Tab manager & memory strategy**

* Each tab holds a PdfViewer instance and an annotation model.
* Implement **lazy page rendering** (render only visible pages, keep rendered canvases for the most recent N pages).
* Implement **tab unloading**: if memory pressure is high or the app has many tabs, unload background tab renderers but keep metadata to quickly reload. Show a lightweight placeholder when unloaded.
* Provide UI for closing/reloading tabs and a tab context menu (Close, Close Others, Duplicate, Detach to Window).

**Build & packaging notes**

* Use electron-builder (or electron-forge) to generate installers for Windows/macOS/Linux.
* When bundling, ensure pdf.worker.js is reachable from your renderer code. If you use asar packaging, either: unpack worker into asarUnpack or load it via BrowserWindow.webContents.executeJavaScript from a blob URL.
* Include pdf-lib and any other libraries under src/libs or node\_modules (bundled). Do not rely on runtime network resources.
* If you later bundle ASP.NET Core backend, publish it as **self-contained** for each platform and add it under resources/ in the packaged installer. Use asarUnpack for executable binaries.

**Testing & QA**

* Unit tests for pure JS modules (annotation serialization, utilities) using **Jest** or **Mocha**.
* E2E tests for UI flows using **Playwright** (supports Electron apps). Avoid Spectron (deprecated).
* Maintain a test-pdfs/ corpus that covers search, forms, outlines, scanned images, and annotations.

**Developer ergonomics & tooling**

* Linting: ESLint with an agreed config and Prettier for formatting.
* Commit hooks: Husky + lint-staged to run format/lint on changed files.
* Source control: adopt conventional commits + CHANGELOG.md generated from commits (optional).
* Documentation: docs/ contains feature specs, API contracts, and onboarding README.

**Starter TODO (first milestones)**

1. Initialize repo structure and add essential packages (pdfjs-dist, pdf-lib).
2. Implement secure preload.js bridge with openFile/readFile/writeFile IPC handlers.
3. Build PdfViewer wrapper that loads a local PDF as ArrayBuffer and renders the first page. Ensure worker is wired locally.
4. Build the simple TabManager UI that creates new tabs on file open and swaps PdfViewer instances.
5. Implement thumbnails sidebar (render small canvases for pages) and outline display.
6. Add basic zoom, fit modes, next/prev page, and keyboard shortcuts.
7. Create test-pdfs/ with representative PDFs and add basic unit tests.

**Appendix — Recommended package.json (minimal)**

{

"name": "tooldeck",

"version": "0.1.0",

"main": "src/main/main.js",

"scripts": {

"start": "electron .",

"dev": "electron . --enable-logging",

"lint": "eslint .",

"test": "jest"

},

"devDependencies": {

"electron": "^38.1.2"

},

"dependencies": {

"pdf-lib": "^1.17.1",

"pdfjs-dist": "^3.9.179"

}

}

If you want, I can now generate:

* a concrete preload.js + main process skeleton (secure) or
* the PdfViewer.js wrapper scaffold (no UI, but a working PDF.js integration) or
* a detailed TabManager UI spec (DOM structure + keyboard mappings)

Tell me which one to produce next and I’ll create it.