Isolated "Ubat Saya" Automation System: A Definitive Architectural Blueprint

Strategic Technical Recommendations and Final Blueprint

This report presents a definitive architectural blueprint for the "Isolated 'Ubat Saya' Automation System," an offline-first application designed for pharmacy terminals. The primary objective is to develop a 100% offline, high-performance application that allows pharmacists to generate visually clear, patient-specific "Ubat Saya" (My Medicines) PDF charts. The system's core is a local, searchable, and *editable* database of medications (ubat), complete with physical descriptions and local image assets.

Based on a rigorous analysis of the project's technical requirements and constraints, a definitive technology stack is recommended. This stack is designed to resolve a cascade of critical, interconnected challenges that render simpler web-based solutions non-viable. The core requirements—100% isolation, a high-performance editable database, high-fidelity PDF generation of a complex CSS layout, and local image asset management—present a significant architectural conflict.

This conflict is most apparent when considering the "Admin Mode" requirement, which allows pharmacists to add new medicines and their corresponding images. In a standard web or Progressive Web App (PWA) environment, a user-provided image cannot be saved to a persistent, same-origin asset folder due to browser sandbox restrictions.[1] The image would be saved as a Blob in IndexedDB.[2] When this Blob is rendered, it is treated as "cross-origin," which subsequently "taints" the HTML canvas.[5] This is a fatal flaw, as the primary library for high-fidelity CSS-to-PDF cloning, html2pdf.js, relies on html2canvas and will fail when encountering a "tainted canvas".[7]

Therefore, the "Admin Mode" image-handling requirement architecturally invalidates a PWA-based approach.

This report's blueprint is built upon a solution that solves this entire cascade:

1. **Platform (Tauri):** A desktop application framework that provides secure, high-performance, native file system access via a Rust backend.[9] This allows user-uploaded images to be saved as *same-origin files*, completely eliminating the "tainted canvas" problem.
2. **Database (Native SQLite):** The choice of Tauri unlocks the ability to bypass slow browser databases (like IndexedDB [11]) and complex WebAssembly (Wasm) implementations.[13] Instead, the system will use native, high-speed SQLite bindings via the tauri-plugin-sql.[15]
3. **Frontend (Svelte):** A lightweight, high-performance compiler that aligns perfectly with the minimal-overhead philosophy of Tauri.[17]
4. **PDF Generation (Hybrid):** A hybrid approach combining html2pdf.js (for visual fidelity of the layout shell) and jspdf-autotable (for accessible, vector-based text data).[19]

The recommended technology stack is, therefore:

- **Platform:** Tauri (Rust/JS)
- **Database:** Native SQLite (via tauri-plugin-sql)
- **Frontend:** Svelte (with SvelteKit)
- **PDF Generation:** html2pdf.js + jspdf-autotable

This architecture is not merely a preference; it is the only solution that robustly and elegantly solves all of the project's stated and implied requirements, delivering the highest performance, lowest resource usage, and greatest long-term stability for a pharmacy terminal environment.

Part 1: Platform Architecture Analysis for an Isolated Pharmacy Terminal

Defining the Deployment Context

The "Ubat Saya" system is specified as a tool for a pharmacy terminal. This context dictates a unique set of non-functional requirements. The application must be:

1. **Instantly Available:** Pharmacists are in a high-throughput, patient-facing role. The application cannot have a long startup time.
2. **Highly Reliable:** It must function 100% offline, as network connectivity in a hospital or

clinic environment is not guaranteed.
3. **Lightweight:** Pharmacy terminals are often not high-specification machines. They may be older Windows-based terminals with limited CPU and RAM. A bloated application that consumes 400MB of RAM on idle is operationally unacceptable.[21]
4. **Stable:** As a "locked-down" utility, it should not be dependent on the user's system browser, which could be updated or changed, breaking a PWA.

The choice of platform—Progressive Web App (PWA), Electron, or Tauri—is the single most important architectural decision and has cascading effects on every other part of the technology stack, from database selection to image handling.

Analysis of Option 1: Progressive Web App (PWA)

A PWA is a web application (HTML/JS/CSS) that can be "installed" on the desktop and run by the user's browser.[23] Its offline capabilities are provided by a Service Worker, which caches all application assets (HTML, CSS, JS, images).[4]

Pros:

- **Low Resource Usage:** Runs within an existing browser tab, leveraging the browser's inherent optimizations.[25]
- **Security:** Operates within the secure browser sandbox, offering a low risk of compromising the user's system.[1]
- **Fast Loading:** Once cached by the service worker, load times are extremely fast.[24]

Cons (Fatal Flaws):
The PWA's primary strength—its security sandbox—is also its greatest weakness and the source of its unsuitability for this project.

- **Constrained File System Access:** A PWA has extremely limited access to the local file system.[1] It cannot directly write a file to a persistent, known location.
- **The "Admin Mode" Failure Cascade:** As outlined in the strategic summary, the requirement for a pharmacist to add a new medicine *with an image* creates an unsolvable problem:
    1. The pharmacist uses <input type="file"> to select an image from the local drive.
    2. The PWA, unable to *copy* this file, must instead read it and store it as a Blob (binary data) in IndexedDB.[2]
    3. To display this image in the PDF chart, the application must read the Blob from IndexedDB and place it in an <img> tag using a blob: URL or a ... URL.
    4. For security reasons, browsers treat these dynamically generated data: or blob: URLs

as "cross-origin" data, even though they originate from the same application.
5. The high-fidelity PDF generator, html2pdf.js, functions by calling html2canvas to take a "screenshot" of the DOM.[8]
6. html2canvas is subject to the "tainted canvas" security rule: it will not export a canvas that has had cross-origin data drawn onto it.[5]
7. Attempting to generate the PDF will fail, throwing a DOMException: Tainted canvases may not be exported.[7]

This conflict between the image-handling requirement and the PDF-generation requirement makes a PWA architecturally non-viable.

Analysis of Option 2: Electron

Electron is a framework for building desktop applications by bundling a full Chromium browser instance and a Node.js runtime.[1]

Pros:

- **Full System Integration:** Electron provides deep, comprehensive access to the operating system.[1]
- **Solves the PWA Flaw:** Using the built-in Node.js fs module, an Electron app can easily read the pharmacist's uploaded image and write it to a local asset folder (e.g., app/assets/images/new_pill.png).[1] This file is now *same-origin*, completely solving the "tainted canvas" problem.

Cons (Fatal Flaws):

- **Resource Consumption and Bloat:** Electron's power comes at a severe cost. It is notoriously "bloated".[31] Because it bundles an *entire browser*, application bundle sizes are massive, often exceeding 100MB.[32]
- **High Memory Usage:** Idle RAM consumption is extremely high, often 200-400MB or more.[21] This is fundamentally unsuitable for the target deployment environment (a simple utility on a pharmacy terminal).

Analysis of Option 3: Tauri

Tauri is a modern framework for building lightweight desktop applications. It consists of a web

technology frontend (HTML/JS/CSS) that communicates with a high-performance **Rust backend**.[29] Crucially, it does *not* bundle Chromium. Instead, it uses the **native system WebView** (e.g., WKWebView on macOS, WebView2 on Windows).[22]

Pros:

- **Minimal Resource Usage:** Tauri apps are "tiny".[29] Because they leverage the system's *existing* WebView, bundle sizes can be under 10MB, and idle RAM usage can be as low as 20-40MB.[21] This is a 90-95% reduction in size and a 5-10x reduction in memory usage compared to Electron.
- **High Performance:** Startup is near-instant, feeling more like a native application.[32]
- **Full System Access (The Solution):** The Rust backend provides a secure, high-performance bridge to the operating system. It has full, native file system access, just like Electron.[9] It can therefore execute the *exact same solution* as Electron (copying the uploaded image to a local, same-origin asset folder) but without any of the bloat.

Cons:

- **UI Inconsistency (Minor):** Using the system's native WebView *can* lead to minor UI rendering differences between, for example, an old Windows 10 machine and a new Windows 11 machine.[35] For a simple, form-based application, this risk is negligible.
- **Rust Learning Curve:** The backend requires Rust.[35] However, for this project's needs (file system and SQL), pre-built plugins (tauri-plugin-fs and tauri-plugin-sql) abstract almost all of the Rust-specific complexity.[9]

Recommendation and Architectural Justification

The analysis reveals a clear and unambiguous winner. PWA fails on the core "Admin Mode" functionality. Electron succeeds on functionality but fails catastrophically on the performance and resource constraints of the deployment environment. Tauri is the only platform that meets *all* requirements: the deep system access of Electron combined with the lightweight performance profile of a PWA.

Table 1: Platform Architecture Trade-Off Analysis

Feature	PWA (Progressive Web App)	Electron	Tauri

Offline Capability	**Excellent** (via Service Worker) [4]	**Excellent** (All files local) [23]	**Excellent** (All files local) [31]
Database Access	Browser-only (IndexedDB / Wasm)	Node.js (e.g., SQLite3) / Wasm	**Native Rust (e.g., SQLite)** [15]
File System Access	**Very Poor** (Sandboxed, no direct write) [1]	**Excellent** (Node.js fs module) [1]	**Excellent** (Rust fs module) [9]
Bundle Size	**Excellent** (Web-based)	**Very Poor** (>100MB) [32]	**Excellent** (<10MB) [21]
Memory Usage	**Excellent** (Browser tab) [25]	**Very Poor** (200MB+ idle) [21]	**Excellent** (20-40MB idle) [21]
"Admin Mode" Suitability	**Poor.** (Fails on image handling)	**Good.** (Functionally capable)	**Excellent.** (Capable and lightweight)
PDF Tainting Risk	**High.** [5]	**Low.** (Same-origin files)	**Zero.** (Same-origin files)

Final Recommendation: Tauri is the optimal and recommended platform. It provides the necessary system-level integration to solve the project's critical image-handling constraints while adhering to the performance and resource requirements of a pharmacy terminal.

Part 2: The Data Layer: Selecting a High-Performance Local Database

The Cascading Decision

The selection of Tauri in Part 1 is a cascading architectural decision that unlocks superior

database options. A PWA-based system would be restricted to browser-based storage technologies. By choosing Tauri, the system gains access to the native operating system, allowing for a much faster, more robust, and simpler database solution. This section will analyze the browser-based options that are *now obsolete* for this project, and contrast them with the recommended native solution.

Analysis of Browser-Based Option 1: IndexedDB (IDB)

IndexedDB (IDB) is a low-level, NoSQL object store built into all modern browsers.[4] It is asynchronous, has a large storage capacity, and is the standard for storing structured data and Blobs in a PWA.[2]

Pros:

- **Asynchronous:** Does not block the main UI thread during database operations.[38]
- **Large Storage:** Can store significantly more data than localStorage.[38]

Cons (Fatal Flaws):

- **Performance:** IndexedDB is notoriously slow. Its performance is known to "fall off a cliff" above a certain cache size.[11] This is because IDB is merely an API specification; the underlying implementation (which could be SQLite or LevelDB) is hidden behind multiple layers of browser abstraction, each adding overhead.[12]
- **Querying:** IDB is *not* a SQL database.[41] The project requires a "search by generic, code, or brand" feature. In IDB, this would require creating multiple indexes and implementing complex, multi-cursor "fuzzy-search" logic.[42] This is slow, difficult to maintain, and far less powerful than a simple SQL LIKE query.
- **Developer Experience:** The native IDB API is complex and cumbersome, leading to a poor developer experience. It is so disliked that "Most developers actively avoid using IndexedDB as much as they can".[41] Workarounds typically involve using a wrapper library like Dexie.js.[44]

Analysis of Browser-Based Option 2: SQLite Wasm (e.g., sql.js, absurd-sql)

This technology involves compiling the entire C-based SQLite library into WebAssembly (Wasm) and running it inside the browser.[46] To achieve persistence, these libraries use

IndexedDB as their underlying storage backend, effectively writing blocks of the database file into an IDB object store.[13]

Pros:

- **Full SQL:** This solution brings the power and familiarity of SQL to the browser, completely solving the querying problem of native IDB.[46]
- **Performance:** Implementations like absurd-sql can be surprisingly fast, even *beating* native IndexedDB performance on benchmarks by optimizing how it writes blocks to the database.[13]

Cons:

- **Complexity and Size:** This approach is an abstraction on top of an abstraction (SQLite-Wasm -> JavaScript Bridge -> IndexedDB -> Native DB).[50] It also requires the user to download a large Wasm binary (409KB - 1.6MB).[49]
- **Startup Overhead:** There is a noticeable startup "tax" required to initialize the Wasm binary and open the database.[14] This conflicts with the <50ms data retrieval goal.

Analysis of Recommended Option: Native SQLite via Tauri Plugin

The Tauri platform, with its Rust backend, allows the application to completely bypass all browser database limitations. The system can interact *directly* with a native SQLite database file (e.g., ubat.db) stored securely in the application's data directory on the user's file system. The tauri-plugin-sql provides all the necessary Rust-to-JS bindings.[15]

Workflow:
This architecture creates a clean, fast, and secure data flow:

1. **Frontend (Svelte):** The user types "Para" into the autocomplete search box.
2. **JS Invoke:** Svelte's JavaScript calls a Tauri command: invoke('search_ubat', { term: 'Para' }).[51]
3. **Backend (Rust):** This triggers a Rust function defined in src-tauri/main.rs: #[tauri::command] fn search_ubat(term: String) ->...
4. **Native Query:** The Rust function, using a native library like rusqlite or sqlx [52], executes a query *directly* against the ubat.db file. This query is simple, standard SQL (e.g., SELECT * FROM ubat WHERE generic_name LIKE?1).
5. **Return:** The Rust function returns the database results as a JSON string, which is deserialized by the Svelte frontend and displayed to the user.

Pros:

- **Performance:** This is the *fastest possible solution*. The query is native code running directly against a native file. It eliminates all overhead from Wasm, JavaScript-to-IDB bridges, and IDB's own abstractions.
- **Simplicity:** The query logic is trivial (a single SQL LIKE statement) compared to the complex cursor-based logic of IDB.[42]
- **Robustness:** The entire database is a single ubat.db file. This file is persistent, portable, and can be easily backed up or versioned, unlike the browser's opaque IndexedDB storage.

Recommendation and Architectural Justification

The choice of Tauri in Part 1 makes the database decision simple. The Native SQLite approach is superior to its browser-based counterparts in every metric: speed, simplicity, and robustness. It provides the power of a true relational database without the performance-tax of Wasm or the-complexity-tax of IndexedDB.

Table 2: Local Database Technology Comparison

Feature	IndexedDB (PWA)	SQLite-Wasm (PWA)	Native SQLite (Tauri)
Query Language	NoSQL (Cursors) [42]	**Full SQL** [46]	**Full SQL (Native)** [52]
Data Retrieval	Slow, high abstraction [11]	Fast (but Wasm overhead) [14]	**Blazing Fast** (Native Rust) [51]
Architecture	JS API -> Browser DB [40]	Wasm -> JS -> IDB [13]	**JS -> Rust -> Native File** [15]
Autocomplete Logic	Complex [43]	Simple (SQL LIKE)	**Trivial** (SQL LIKE)
Data Persistence	Browser Storage (IDB)	Browser Storage (IDB) [49]	**Local .db File** [15]

Final Recommendation: The Tauri-Native-SQLite stack is the clear and logical choice. It delivers the best performance, the simplest query logic, and the most robust data persistence

model, all directly enabled by the platform architecture.

Part 3: Frontend Framework and UI/UX Implementation

The Role of the Frontend

With the platform (Tauri) and database (Native SQLite) selected, the final piece of the core stack is the frontend framework. The framework will be responsible for rendering the UI (the forms and the PDF chart preview), managing the application's state (which medicines have been added for the patient), and calling the Tauri Rust backend via invoke commands to search for and manage ubat data.

The key consideration is *philosophical alignment*. The primary driver for choosing Tauri over Electron was its lightweight, high-performance, minimal-overhead nature. It would be architecturally inconsistent to then select a heavy, "bloated" frontend framework that negates these advantages.[55]

Analysis of Option 1: React

React is a UI *library* with a massive, flexible ecosystem.[56] It uses a "virtual DOM" to manage UI updates.[18]

Pros:

- **Ecosystem:** Has the largest community and job market.[55]
- **Flexibility:** As a library, it is unopinionated, allowing for a wide range of state management and routing solutions.

Cons:

- **Overhead:** The "virtual DOM" concept introduces a layer of abstraction and runtime overhead that is unnecessary for a simple application.[18]

- **Boilerplate:** Generally requires more boilerplate code for setup and state management compared to alternatives.[57]
- **Ecosystem Bloat:** Its "massive but fragmented" ecosystem can be a drawback, requiring developers to pull in many dependencies to build a full application.[55]

Analysis of Option 2: Vue

Vue is a "progressive framework" that offers a more cohesive, beginner-friendly experience.[56] It is often described as an "elegant middle ground" between the complexity of React and the simplicity of Svelte.[57]

Pros:

- **Learning Curve:** Widely considered the easiest to pick up for beginners.[18]
- **Balance:** Offers a good balance of performance and flexibility.[18] Its reactivity system and virtual DOM are well-optimized.

Cons:

- **Verbose Syntax:** Some developers find its HTML-based syntax (v-for, @click, <template>) to be verbose or "unattractive".[55]
- **Overhead:** Like React, it is still a framework that ships a runtime and a virtual DOM to the browser, adding overhead that a compiler-based approach avoids.

Analysis of Option 3: Svelte

Svelte is a radical new approach. It is not a library or framework; it is a **compiler** that runs at *build time*.[17] It takes declarative Svelte components and compiles them into highly efficient, minimal JavaScript that *surgically updates the DOM directly*.[45]

Pros:

- **Performance:** Svelte is the "new performance champion".[18] Because it does *not* use a virtual DOM, its runtime performance is "extremely fast" and "best in class".[17]
- **Minimal Bundle Size:** It ships the *least possible overhead*. The final JavaScript bundle contains only the minimal code necessary to run the application, not a large framework runtime.[17]
- **Simplicity:** The code is "by far the cleanest".[57] It requires "less boilerplate" and has a

"simple syntax" [17], as reactivity is built into the language itself.
- **Architectural Fit:** Svelte's philosophy of "minimal overhead" is a *perfect* technical and philosophical match for Tauri's lightweight platform. This combination (Tauri + Svelte) produces the smallest, fastest, and most resource-efficient desktop application possible.[58]

Cons:

- **Smaller Ecosystem:** Its community is smaller and newer compared to React's.[17] This is not a significant risk for a small, isolated utility application that does not require a vast ecosystem of third-party plugins.

Recommendation and Architectural Justification

Svelte is the logical and superior choice for this project. Its compiler-first, no-virtual-DOM approach delivers the fastest runtime and smallest bundle size, perfectly complementing the lightweight, high-performance architecture of Tauri. This ensures the final application remains fast, lightweight, and responsive, even on older pharmacy terminals.

Table 3: Frontend Framework Selection Matrix

Framework	Paradigm	Performance	Bundle Size	Learning Curve	Tauri Integration
React	UI Library (Virtual DOM) [18]	Good	Large [55]	Moderate [18]	Good [60]
Vue	Progressive Framework [56]	Good [18]	Medium	Easy [18]	Good [61]
Svelte	**Compiler** (No V-DOM) [17]	**Excellent** [18]	**Minimal** [17]	Easy [18]	**Excellent** [58]

Final Recommendation: Svelte. The combination of Tauri, Native SQLite, and Svelte represents a "best-in-class" stack for lightweight, high-performance, data-driven desktop applications.

Part 4: Core Functionality Deep Dive

This section details the implementation of the system's core features, based on the recommended Tauri + Native SQLite + Svelte architecture.

Finalized Ubat (Medicine) Database Schema

The central data store will be a native SQLite file (ubat.db). The schema is designed for fast, flexible searching and to store all necessary data for auto-population of the PDF chart. The brand_variants column will store a JSON-formatted text array, allowing a single LIKE query to search this "many-to-one" relationship.

The data in the example columns is derived from the provided medical and pharmaceutical sources, demonstrating how the database will be populated with real-world, patient-friendly information.

Table 4: Final Ubat (Medicine) Database Schema

Column	Type	Description	Example 1 (Gliclazide)	Example 2 (Perindopril)	Example 3 (Amlodipine)	Example 4 (Atorvastatin)	Example 5 (Pantoprazole)	Example 6 (Metoprolol)
code	TEXT (PRIMARY KEY)	Internal pharmacist reference.	GLI30MR	PERIN8	AMLO5	ATORV20	PANTO40	METO50
generic_name	TEXT	The full generi	Gliclazide 30mg	Perindopril Erbum	Amlodipine Besyla	Atorvastatin 20mg	Pantoprazole	Metoprolol Tartrat

		c name and dosage.	MR	ine 8mg	te 5mg		40mg	e 50mg
brand_variants	TEXT (JSON Array)	Common brand names.	`` [62]	["Perinace", "Coversyl", "Aceon"] [64]	["Norvasc", "Covasc"] [66]	`` [69]	["Vencid", "Protonix", "Nolpaza"] [71]	["Lopressor"] [73]
form	TEXT	e.g., Tablet, Capsule, Syrup.	Tablet	Tablet	Tablet	Tablet	Tablet	Tablet
shape	TEXT	Physical shape description.	Oval / Capsule-shaped [62]	Round [64]	Eight-sided [66]	Oval [75]	Oval [77]	Round [79]
color	TEXT	Physical color description.	White [62]	White [64]	White [66]	White [75]	Yellow [77]	Peach / Pink [79]
image	TEXT	**Local relative file path.**	gli30 mr.png	perin8.png	amlo5.png	atorv20.png	panto40.png	meto50.png
instruction	TEXT	Default patient	"Ambil sekali sehari denga	"Ambil sekali sehari, sebelu	"Ambil sekali sehari, pada	"Ambil sekali sehari, bila-bi	"Ambil 30 minit sebelu	"Ambil dengan makan

		instruction.	n sarapan. Telan seluruhnya." [81]	m sarapan." [83]	masa yang sama setiap hari." [86]	la masa, dengan atau tanpa makanan." [88]	m sarapan." [89]	an atau sejurus selepas makan." [91]

Implementing High-Speed Autocomplete

The autocomplete feature is critical for the pharmacist's workflow. The Tauri/SQLite stack makes this implementation trivial and extremely fast.

UI (Svelte):
A Svelte-based autocomplete component, such as Svelecte 93 or simple-svelte-autocomplete 95, will be used. These components support asynchronous search functions.
Data Flow:

1. **Input:** The pharmacist types "Para" into the Svelecte component.
2. **Event:** The component's searchFunction [95] or equivalent is triggered.
3. **Frontend (JS):** This JavaScript function imports the invoke function from the Tauri API: import { invoke } from '@tauri-apps/api/tauri'.
4. **Tauri Command:** It executes the command, passing the search term: let results = await invoke('search_db', { term: 'para%' });. The % is a SQL wildcard.
5. **Backend (Rust):** A Rust function in src-tauri/main.rs is registered to handle this command: #[tauri::command] fn search_db(state: tauri::State<DbState>, term: String) -> Result<Vec<Ubat>, String>. This function accesses the thread-safe SQLite connection pool.[60]
6. **SQL Query:** The Rust function executes a single, powerful SQL query:
   ```
   SQL
   SELECT * FROM ubat
   WHERE code LIKE?1
   OR generic_name LIKE?1
   OR brand_variants LIKE?1
   LIMIT 10;
   ```
   This query instantly searches the primary key, the generic name, and the *contents* of the JSON brand list for the search term.
7. **Response:** The query results are returned as a JSON Vec<Ubat> to the Svelte frontend,

which populates the autocomplete dropdown.

This architecture is vastly superior to the PWA/IndexedDB alternative, which would require a complex, non-standard "starts-with" query using IDBKeyRange.bound(searchTerm, searchTerm + '\uffff').[42] The Tauri/SQL LIKE operator is simpler, more flexible, and more powerful.

Solving the Image Problem (Offline Management & "Admin Mode")

This section details the robust solution for the "Admin Mode" image-handling, which was the primary driver for the Tauri platform selection.

Storage Location:
All medicine images (e.g., amlo5.png, perin8.png) will be stored in a local application data directory, not in a database. Tauri provides a cross-platform API to access this directory, such as BaseDirectory.LocalData.36 The file path (e.g., assets/img/ubat/amlo5.png) will be stored in the SQLite database's image column (see Table 4).
"Admin Mode" - Adding a New Image:
This workflow demonstrates the power of the Tauri architecture.

1. **UI (Svelte):** The pharmacist navigates to the /admin page and fills out a form for a new medicine (e.g., Code: NEW100, Name: New Ubat 100mg).
2. **File Input:** They use a standard <input type="file" accept="image/*"> to select the pill image from their computer.
3. **Frontend (JS):** An on:change event handler reads the selected file object. It does *not* create a Base64 string. Instead, it reads the file as a Uint8Array (a binary byte array).
4. **Tauri Command:** The frontend invokes a single command to the Rust backend, passing both the text details and the binary image data:

```JavaScript
// (Inside Svelte component)
import { invoke } from '@tauri-apps/api/tauri';

async function handleSave(medicineDetails, imageFile) {
  const imageBytes = new Uint8Array(await imageFile.arrayBuffer());

  await invoke('add_new_medicine', {
    details: medicineDetails, // { code: 'NEW100',... }
    image_bytes: Array.from(imageBytes) // Pass binary data
  });
}
```

5. Backend (Rust): The add_new_medicine Rust command executes a two-step, transactional process:
   a. Database Write: It runs an INSERT INTO ubat (...) command using the details object.
   b. File System Write: It uses the tauri-plugin-fs to save the binary data as a new file in the application's same-origin asset directory.9 For example: writeBinaryFile("assets/img/ubat/NEW100.png", image_bytes, { dir: BaseDirectory.LocalData }).36
6. **Result:** The new medicine is in the database, and its image is a local, same-origin file. When this image is rendered in the UI, it will *not* taint the canvas.

This pattern is infinitely more robust than the PWA alternative of storing image Blobs in IndexedDB.[2] It is simpler to manage, faster to read, and, most importantly, it completely preempts the html2canvas security error, ensuring the PDF generation feature works 100% of the time.

Part 5: Achieving High-Fidelity PDF Generation

The Fidelity vs. Functionality Dilemma

The project has two conflicting PDF requirements.

1. **High Fidelity:** The output must have >= 90% visual fidelity to the *colorful, CSS-heavy* sample layout.
2. **Accessibility:** The output should be a *readable* document for patients, which implies selectable text (e.g., for copy-pasting a medicine name).

Standard JavaScript PDF libraries force a choice between these two goals.

Analysis of Option 1: pdfmake

pdfmake is a library for creating PDF documents from a declarative, JSON-based definition.[96]

Pros:

- **True PDF:** It generates a *true* PDF document with selectable, searchable, vector-based text.[19]
- **Structured Data:** It is excellent for structured layouts like invoices, reports, and tables.[98]

Cons (Fatal Flaw):

- **No HTML/CSS Cloning:** pdfmake *cannot* read a DOM element and clone its style. It has "no ability for overlapping elements or positioned elements".[99] To meet the fidelity requirement, the developer would have to manually *re-create* the entire colorful layout using pdfmake's JSON syntax. This is extremely brittle, time-consuming, and will never achieve 100% fidelity. It fails the high-fidelity requirement.

Analysis of Option 2: html2pdf.js

html2pdf.js is a popular library that works as a wrapper around two other libraries: html2canvas and jspdf.[8] Its process is to "capture the exact look of your web page".[8]

Pros:

- **Perfect Fidelity:** It is *designed* to clone complex HTML and CSS. It will capture the pastel colors, borders, shadows, and icons of the layout *exactly* as they appear on the screen. It perfectly meets the high-fidelity requirement.

Cons:

- **Non-Selectable Text:** The output is an *image* of the DOM, which is then placed on a PDF page. This means the text in the PDF is not selectable, searchable, or accessible.[19] This fails the accessibility requirement.
- **Tainted Canvas Vulnerability:** As previously established, html2canvas will fail if any "cross-origin" images (like those loaded from IndexedDB Blobs) are present.[5] (This specific vulnerability has already been mitigated by the Tauri platform choice, but the non-selectable text issue remains).

Recommendation: The Hybrid "Best-of-Both" Solution

Neither library alone can satisfy all project requirements. pdfmake fails fidelity, and html2pdf.js fails accessibility.

However, a superior, hybrid solution exists. html2pdf.js is built on jspdf. The jspdf library, in

turn, has a powerful plugin for creating structured tables: jspdf-autotable.[8] This allows for a multi-stage generation process that combines the strengths of both methods.

The Hybrid Workflow:

1. **Prepare UI:** The Svelte UI is already styled to look *exactly* like the target PDF. When the "Generate PDF" button is clicked, the application *temporarily hides* the dynamic medicine rows (e.g., visibility: hidden).
2. **Generate Shell (Fidelity):** html2pdf.js is run on the *main application shell*—the component containing the header, footer, colorful borders, and the time-of-day icons (🌅, ☀️, 🌇, 🌙).
3. **Capture jspdf Instance:** This process creates a new jspdf document instance with the beautiful, high-fidelity shell rendered as a *background image*. The promise from html2pdf.js provides access to this jspdf object.
4. **Generate Data (Accessibility):** The application *does not* save the PDF yet. Instead, it now uses the jspdf-autotable plugin [20] to draw a *new, vector-based table* (with *real, selectable text*) directly *on top* of the background image. The data for this table is pulled directly from the Svelte application's state.
5. **Final Output:** The final doc.save('Ubat_Saya_[NamaPesakit].pdf') command is called. The result is a single PDF that has the pixel-perfect CSS layout from html2canvas *and* the accessible, selectable text from jspdf-autotable.

This hybrid approach is the optimal solution, as it is the only one that satisfies both the >= 90% visual fidelity and the patient-accessibility requirements.

Table 5: PDF Generation Library Feature Comparison

Library	Method	Visual Fidelity (HTML Cloning)	Selectable Text	Key Trade-Off
pdfmake	Declarative (JSON) [98]	**Very Poor** [99]	**Excellent** [19]	Fails the visual fidelity requirement.
html2pdf.js	DOM "Screenshot" [8]	**Excellent** [8]	**Very Poor** [19]	Fails the accessibility requirement.
Hybrid (Recommend	**Hybrid** (Image Shell + Vector	**Excellent**	**Excellent**	Solves all requirements.

ed)	Table)			

Part 6: Final System Specification and Implementation Roadmap

Consolidated Technology Stack Blueprint

This blueprint consolidates all recommendations into a final, coherent technology stack designed for maximum performance, stability, and adherence to all project constraints.

- **Platform:** Tauri (for a lightweight, native-WebView desktop app) [29]
- **Backend:** Rust (for high-performance system-level tasks) [35]
- **Frontend:** Svelte/SvelteKit (for a fast, minimal-overhead, compiler-based UI) [17]
- **Database:** Native SQLite (for high-speed, robust, local-file-based data) [15]
- **Database Plugin:** tauri-plugin-sql (to bridge Rust/SQLite to the frontend) [15]
- **File Management:** tauri-plugin-fs (for native image file saving) [9]
- **UI Components:** Svelecte (or similar, for asynchronous autocomplete) [93]
- **PDF Generation:** html2pdf.js (for CSS shell fidelity) + jspdf-autotable (for accessible data) [19]

Proposed File Structure

The recommended project structure, based on the SvelteKit/Tauri template, is as follows:

/src/

| /lib/
| | /components/

```
| | | - MedicineRow.svelte
| | | - AutocompleteUbat.svelte
| | - db.ts              (JS helper functions to call Tauri `invoke` commands)
| /routes/
| | - +page.svelte       (The main "Ubat Saya" generation form)
| | - +layout.svelte     (The main application shell/layout)
| | /admin/
| | | - +page.svelte    (The "Admin Mode" form for adding/editing ubat)
/src-tauri/

| /migrations/           (SQL schema definitions)
| /src/
| | - main.rs           (Rust main entry point, DB setup, `#[tauri::command]` definitions)
| | - db.rs             (Rust module for all native SQLite query logic)
| - Cargo.toml          (Rust dependencies: tauri, tauri-plugin-sql, tauri-plugin-fs, rusqlite)
| - tauri.conf.json     (Tauri config: allowlist for fs and sql plugins)
/appdata/ (Managed by Tauri, in user's OS-specific data directory)

| - ubat.db             (The native SQLite database file)
| /assets/
| | /img/
| | | /ubat/
| | | | - amlo5.png
| | | | - panto40.png
| | | | - (newly added images will be saved here)
```

Step-by-Step Development Roadmap

1. **Phase 1: Environment Setup (Tauri + Svelte + SQLite)**
    - Initialize a new SvelteKit project.
    - Add Tauri to the project using the CLI: npm run tauri init.[58]
    - Add the required Rust dependencies to Cargo.toml: cargo add tauri-plugin-sql --features sqlite and cargo add tauri-plugin-fs.[16]
    - Configure tauri.conf.json to grant permissions ("allowlist") for the fs and sql plugins to access the application data directory.[36]
    - Write the initial Rust function in main.rs to initialize and connect to the ubat.db file (using Database.load("sqlite:ubat.db")) on application startup.[15]
2. **Phase 2: Database Seeding**
    - Create a one-time Rust function to be called on first launch.

- This function will execute CREATE TABLE ubat (...) based on the schema in Table 4.
- It will then run INSERT statements to populate the database with the initial 10-20 most common medicines, using the data gathered from pharmaceutical sources.[72]

3. **Phase 3: Core UI & Autocomplete**
   - Build the main Svelte UI (+page.svelte) with the patient name input and dynamic, addable/removable medicine rows.
   - Integrate the Svelecte autocomplete component.[93]
   - Implement the #[tauri::command] fn search_db(...) in main.rs as described in section 4.2.[51]
   - Wire the Svelecte component's search function to invoke the search_db command, completing the "Svelte -> Rust -> SQLite -> Svelte" data loop.
   - On selection, auto-populate the row with the data (image, name, instructions).
4. **Phase 4: Admin Mode (Image & Data Management)**
   - Build the /admin Svelte route with a form for adding/editing ubat records.
   - Implement the #[tauri::command] fn add_new_medicine(...) in main.rs.
   - This function must:
     a. Accept the text details and the binary image data (Vec<u8>).
     b. Execute the INSERT or UPDATE SQL command.
     c. Use the tauri-plugin-fs writeBinaryFile function to save the image to the appdata/assets/img/ubat/ directory, using the code as the filename (e.g., NEW100.png).9
5. **Phase 5: PDF Generation**
   - Implement the Hybrid PDF strategy from Part 5.
   - Create a JavaScript function generatePdf(patientName, medicineList).
   - This function will:
     a. Temporarily hide the dynamic data rows in the Svelte UI.
     b. Call html2pdf.js on the static UI shell to get the jspdf object.
     c. Use jspdf-autotable to draw the medicineList data (with selectable text) onto the jspdf object at the correct X/Y coordinates.20
     d. Call doc.save('Ubat_Saya_[NamaPesakit].pdf').
6. **Phase 6: Packaging and Deployment**
   - Run the final build command: npm run tauri build.
   - Tauri will compile the Svelte frontend, compile the Rust backend, and bundle them into a single, lightweight, standalone installer (e.g., a .msi file for Windows) for simple distribution to pharmacy terminals.

Works cited

1. Progressive Web Apps (PWAs) vs. Electron Apps — File System Access and Risk Monitoring, accessed November 12, 2025, https://labs.sqrx.com/progressive-web-apps-pwas-vs-electron-apps-file-system-access-and-risk-monitoring-08a496a2d6b3
2. Storing images and files in IndexedDB - Robert's talk, accessed November 12,

2025, https://robertnyman.com/2012/03/06/storing-images-and-files-in-indexeddb/
3. Storing (and Retrieving) Photos in IndexedDB - Raymond Camden, accessed November 12, 2025, https://www.raymondcamden.com/2018/10/05/storing-retrieving-photos-in-indexeddb
4. Offline data - PWA - web.dev, accessed November 12, 2025, https://web.dev/learn/pwa/offline-data
5. Use cross-origin images in a canvas - HTML - MDN Web Docs, accessed November 12, 2025, https://developer.mozilla.org/en-US/docs/Web/HTML/How_to/CORS_enabled_image
6. Tainted canvases may not be exported - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/22710627/tainted-canvases-may-not-be-exported
7. How can I solve the "Tainted canvas" error caused by html2canvas? - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/74906486/how-can-i-solve-the-tainted-canvas-error-caused-by-html2canvas
8. Generate PDFs with jsPDF: A Complete Guide to Client-Side PDF Creation - PDFBolt, accessed November 12, 2025, https://pdfbolt.com/blog/generate-html-to-pdf-with-jspdf
9. File System - Tauri, accessed November 12, 2025, https://v2.tauri.app/plugin/file-system/
10. How to save file from image input? · tauri-apps tauri · Discussion #5450 - GitHub, accessed November 12, 2025, https://github.com/tauri-apps/tauri/discussions/5450
11. How we sped up Notion in the browser with WASM SQLite : r/programming - Reddit, accessed November 12, 2025, https://www.reddit.com/r/programming/comments/1f325me/how_we_sped_up_notion_in_the_browser_with_wasm/
12. Why IndexedDB is slow and what to use instead : r/javascript - Reddit, accessed November 12, 2025, https://www.reddit.com/r/javascript/comments/r0axv1/why_indexeddb_is_slow_and_what_to_use_instead/
13. The Current State Of SQLite Persistence On The Web: November 2025 Update - PowerSync, accessed November 12, 2025, https://www.powersync.com/blog/sqlite-persistence-on-the-web
14. How we sped up Notion in the browser with WASM SQLite - Hacker News, accessed November 12, 2025, https://news.ycombinator.com/item?id=40949489
15. SQL - Tauri, accessed November 12, 2025, https://v2.tauri.app/plugin/sql/
16. FocusCookie/tauri-sqlite-example: This repository provides a guide on setting up an SQLite database with a Tauri 2.0 app and interacting with it from the frontend (JavaScript/TypeScript). It covers backend SQLite configuration and exposing

methods to the frontend, enabling seamless database operations like storing, retrieving, and managing local - GitHub, accessed November 12, 2025, https://github.com/FocusCookie/tauri-sqlite-example
17. Which Progressive Web App Framework Is Best for Your Next Project? - Seven Square, accessed November 12, 2025, https://www.sevensquaretech.com/which-pwa-framework-best-for-your-project/
18. JavaScript Frameworks in 2024: React vs. Vue vs. Svelte – Which One to Choose?, accessed November 12, 2025, https://dev.to/tarunsinghofficial/javascript-frameworks-in-2024-react-vs-vue-vs-svelte-which-one-to-choose-4c0p
19. Convert HTML to PDF with JavaScript: Top libraries compared - Nutrient SDK, accessed November 12, 2025, https://www.nutrient.io/blog/html-to-pdf-in-javascript/
20. Generate Professional & Beautiful PDF Tables with jsPDF-Table - CSS Script, accessed November 12, 2025, https://www.cssscript.com/pdf-tables-jspdf/
21. Electron vs Tauri: Choosing the Best Framework for Desktop Apps - SoftwareLogic, accessed November 12, 2025, https://softwarelogic.co/en/blog/how-to-choose-electron-or-tauri-for-modern-desktop-apps
22. Tauri vs. Electron: performance, bundle size, and the real trade-offs - Hopp, accessed November 12, 2025, https://www.gethopp.app/blog/tauri-vs-electron
23. Progressive web app(PWA) vs Electron vs Browser extension - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/50705226/progressive-web-apppwa-vs-electron-vs-browser-extension
24. Magento PWA vs Electron: Best Framework for Your Ecommerce Needs - MGT Commerce, accessed November 12, 2025, https://www.mgt-commerce.com/blog/magento-pwa-vs-electron/
25. PWA vs Electron: Which Architecture Should You Choose? - Magenest, accessed November 12, 2025, https://magenest.com/en/pwa-vs-electron/
26. Using IndexedDB to save images - javascript - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/22740186/using-indexeddb-to-save-images
27. snowball-tools/image-upload-pwa-example - GitHub, accessed November 12, 2025, https://github.com/snowball-tools/image-upload-pwa-example
28. Building An Offline-Friendly Image Upload System - Smashing Magazine, accessed November 12, 2025, https://www.smashingmagazine.com/2025/04/building-offline-friendly-image-upload-system/
29. PWA, IWA, Tauri: The Future of Web-based App Deployment?, accessed November 12, 2025, https://javascript-conference.com/general-web-development/web-based-app-deployment/
30. PWA vs Electron - Which Architecture Wins? - Clean Commit, accessed

November 12, 2025, https://cleancommit.io/blog/pwa-vs-electron-which-architecture-wins/
31. PWA or Electron with React - Reddit, accessed November 12, 2025, https://www.reddit.com/r/react/comments/14bluzt/pwa_or_electron_with_react/
32. Tauri vs Electron Comparison: Choose the Right Framework | by RaftLabs | Sep, 2025, accessed November 12, 2025, https://raftlabs.medium.com/tauri-vs-electron-a-practical-guide-to-picking-the-right-framework-5df80e360f26
33. Tauri vs. Electron Benchmark: ~58% Less Memory, ~96% Smaller Bundle – Our Findings and Why We Chose Tauri : r/programming - Reddit, accessed November 12, 2025, https://www.reddit.com/r/programming/comments/1jwjw7b/tauri_vs_electron_benchmark_58_less_memory_96/
34. Tauri vs Electron JS for desktop apps - amazing memory & disk usage - YouTube, accessed November 12, 2025, https://www.youtube.com/watch?v=BQrtEglqqGg
35. [AskJS] Tauri vs Electron : r/javascript - Reddit, accessed November 12, 2025, https://www.reddit.com/r/javascript/comments/ulpeea/askjs_tauri_vs_electron/
36. With Tauri how can i save a file to a specific path? - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/78221816/with-tauri-how-can-i-save-a-file-to-a-specific-path
37. Tauri Development - Saving Files - YouTube, accessed November 12, 2025, https://www.youtube.com/watch?v=WDPZb_zVrd8
38. How to Use IndexedDB for Data Storage in PWAs, accessed November 12, 2025, https://blog.pixelfreestudio.com/how-to-use-indexeddb-for-data-storage-in-pwas/
39. Using IndexedDB - Web APIs - MDN Web Docs, accessed November 12, 2025, https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB
40. IndexedDB Performance and IndexedDB v/s WebSQL performance comparison - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/29220099/indexeddb-performance-and-indexeddb-v-s-websql-performance-comparison
41. html - Offline access - SQLite or Indexed DB? - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/12247380/offline-access-sqlite-or-indexed-db
42. IndexedDB Fuzzy Search - javascript - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/7086180/indexeddb-fuzzy-search
43. Databinding UI elements with IndexedDB | Articles - web.dev, accessed November 12, 2025, https://web.dev/articles/indexeddb-uidatabinding
44. Using Web Browser's Indexed DB in SvelteKit - DEV Community, accessed November 12, 2025, https://dev.to/theether0/using-web-browsers-indexed-db-in-sveltekit-3oo3
45. IndexedDB for Local Data Storage in a Svelte Project - Thoughts? : r/sveltejs -

Reddit, accessed November 12, 2025, https://www.reddit.com/r/sveltejs/comments/18v139r/indexeddb_for_local_data_storage_in_a_svelte/
46. sql-js/sql.js: A javascript library to run SQLite on the web. - GitHub, accessed November 12, 2025, https://github.com/sql-js/sql.js/
47. SQLite Wasm in the browser backed by the Origin Private File System | Blog, accessed November 12, 2025, https://developer.chrome.com/blog/sqlite-wasm-in-the-browser-backed-by-the-origin-private-file-system
48. sqlite3 wasm docs: sqlite3 WebAssembly & JavaScript Documentation Index, accessed November 12, 2025, https://sqlite.org/wasm
49. npiesco/absurder-sql - GitHub, accessed November 12, 2025, https://github.com/npiesco/absurder-sql
50. SQLite On The Web: Absurd-sql | Hackaday, accessed November 12, 2025, https://hackaday.com/2021/08/24/sqlite-on-the-web-absurd-sql/
51. How Create SQLite database in a tauri TS app? - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/75760019/how-create-sqlite-database-in-a-tauri-ts-app
52. Tauri 2.0 alpha, Svelte, SvelteFlow and SQLite - Gatewaynode, accessed November 12, 2025, https://gatewaynode.com/tauri-20-alpha-svelte-svelteflow-and-sqlite
53. Embedding a SQLite database in a Tauri Application : r/learnrust - Reddit, accessed November 12, 2025, https://www.reddit.com/r/learnrust/comments/1hrsq8v/embedding_a_sqlite_database_in_a_tauri_application/
54. How to speed up performance of autocomplete from indexeddb database - Stack Overflow, accessed November 12, 2025, https://stackoverflow.com/questions/62746611/how-to-speed-up-performance-of-autocomplete-from-indexeddb-database
55. Why Choose Svelte Over Vue or React? : r/sveltejs - Reddit, accessed November 12, 2025, https://www.reddit.com/r/sveltejs/comments/1jy4m01/why_choose_svelte_over_vue_or_react/
56. React vs. Vue vs. Svelte: The Real 2025 Guide to Picking Your First JavaScript Framework, accessed November 12, 2025, https://clinkitsolutions.com/react-vs-vue-vs-svelte-the-real-2025-guide-to-picking-your-first-javascript-framework/
57. React vs. Vue vs. Svelte: The 2025 Performance Comparison | by Jessica Bennett - Medium, accessed November 12, 2025, https://medium.com/@jessicajournal/react-vs-vue-vs-svelte-the-ultimate-2025-frontend-performance-comparison-5b5ce68614e2
58. A basic template for making cross-platform apps with Tauri (Rust), SvelteKit, and SQLite (SQLx) - GitHub, accessed November 12, 2025, https://github.com/Lmedmo/Tauri-SvelteKit-SQLite

59. Offline App with SvelteKit + SQLite Part 1: Setup WebAssembly SQLite - YouTube, accessed November 12, 2025, https://www.youtube.com/watch?v=Uvnzwp72Ze8
60. SQLite Database on Tauri, React, Typescript : r/tauri - Reddit, accessed November 12, 2025, https://www.reddit.com/r/tauri/comments/1iy8989/sqlite_database_on_tauri_react_typescript/
61. Introducing Svelte, and Comparing Svelte with React and Vue (2021) | Hacker News, accessed November 12, 2025, https://news.ycombinator.com/item?id=32763509
62. APO-GLICLAZIDE MR (Gliclazide Modified Release Tablets) Page 1 of 46 PRODUCT MONOGRAPH INCLUDING PATIENT MEDICATION INFORMATION, accessed November 12, 2025, https://pdf.hres.ca/dpd_pm/00075540.PDF
63. Remicron MR | 30 mg | Tablet | রেমিক্রন এমআর ৩০ মি.গ্রা. ট্যাবলেট - MedEx, accessed November 12, 2025, https://medex.com.bd/brands/35425/remicron-mr-30-mg-tablet
64. D 5 9 Pill White Round 8mm - Pill Identifier - Drugs.com, accessed November 12, 2025, https://www.drugs.com/imprints/d-5-9-15869.html
65. Perindopril - Wikipedia, accessed November 12, 2025, https://en.wikipedia.org/wiki/Perindopril
66. Amlodipine Besylate Pictures & Common Dosing, accessed November 12, 2025, https://www.wellrx.com/amlodipine-besylate/drug-images/
67. NORVASC 5 Pill White Eight-sided - Pill Identifier - Drugs.com, accessed November 12, 2025, https://www.drugs.com/imprints/norvasc-5-29429.html
68. Norvasc Pill Images - What does Norvasc look like? - Drugs.com, accessed November 12, 2025, https://www.drugs.com/image/norvasc-images.html
69. 61 Atorvastatin Stock Photos, High-Res Pictures, and Images - Getty Images | Lipitor, accessed November 12, 2025, https://www.gettyimages.com/photos/atorvastatin
70. Storvas 20 Tablet: View Uses, Side Effects, Price and Substitutes | 1mg, accessed November 12, 2025, https://www.1mg.com/drugs/storvas-20-tablet-70629
71. 2+ Hundred Pantoprazole Royalty-Free Images, Stock Photos & Pictures | Shutterstock, accessed November 12, 2025, https://www.shutterstock.com/search/pantoprazole
72. Vencid 40mg Tablet: View Uses, Side Effects, Price and Substitutes | 1mg, accessed November 12, 2025, https://www.1mg.com/drugs/vencid-40mg-tablet-324134
73. Metoprolol Tartrate Pictures & Common Dosing - ScriptSave WellRx, accessed November 12, 2025, https://www.wellrx.com/metoprolol-tartrate/drug-images/
74. Perindopril Pill Images - What does perindopril look like? - Drugs.com, accessed November 12, 2025, https://www.drugs.com/image/perindopril-images.html
75. Atorvastatin Pill Images - Pill Identifier - Drugs.com, accessed November 12, 2025, https://www.drugs.com/imprints.php?drugname=atorvastatin
76. Atorvastatin Pill Images - What does atorvastatin look like? - Drugs.com, accessed November 12, 2025, https://www.drugs.com/image/atorvastatin-images.html

77. Pantoprazole Sodium Pictures & Common Dosing - ScriptSave WellRx, accessed November 12, 2025, https://www.wellrx.com/pantoprazole-sodium/drug-images/
78. Pantoprazole Pill Images - What does pantoprazole look like? - Drugs.com, accessed November 12, 2025, https://www.drugs.com/image/pantoprazole-images.html
79. Metoprolol Pill Images - Pill Identifier - Drugs.com, accessed November 12, 2025, https://www.drugs.com/imprints.php?drugname=metoprolol&maxrows=18
80. Metoprolol tartrate 50 mg film-coated tablets, accessed November 12, 2025, https://www.medicines.org.uk/emc/files/pil.5200.pdf
81. How and when to take gliclazide - NHS, accessed November 12, 2025, https://www.nhs.uk/medicines/gliclazide/how-and-when-to-take-gliclazide/
82. Consumer Information for: MYLAN-GLICLAZIDE MR - Drug and Health Products Portal, accessed November 12, 2025, https://dhpp.hpfb-dgpsa.ca/dhpp/resource/92302/consumer-information
83. How and when to take perindopril - NHS, accessed November 12, 2025, https://www.nhs.uk/medicines/perindopril/how-and-when-to-take-perindopril/
84. Pms Perindopril 8mg tablet – – Medication guide - Familiprix, accessed November 12, 2025, https://www.familiprix.com/en/medications/pms-perindopril-8mg-tablet-02470691
85. PACKAGE LEAFLET: INFORMATION FOR THE PATIENT Perindopril 2mg, 4mg & 8mg Tablets Perindopril, accessed November 12, 2025, https://www.medicines.org.uk/emc/files/pil.11527.pdf
86. Amlodipine (oral route) - Side effects & dosage - Mayo Clinic, accessed November 12, 2025, https://www.mayoclinic.org/drugs-supplements/amlodipine-oral-route/description/drg-20061784
87. How and when to take amlodipine - NHS, accessed November 12, 2025, https://www.nhs.uk/medicines/amlodipine/how-and-when-to-take-amlodipine/
88. Label: ATORVASTATIN CALCIUM tablet, film coated - DailyMed - NIH, accessed November 12, 2025, https://dailymed.nlm.nih.gov/dailymed/lookup.cfm?setid=86841382-4229-4e03-958e-3ac22639efd4
89. Pantoprazole (oral route) - Side effects & dosage - Mayo Clinic, accessed November 12, 2025, https://www.mayoclinic.org/drugs-supplements/pantoprazole-oral-route/description/drg-20071434
90. How and when to take pantoprazole - NHS, accessed November 12, 2025, https://www.nhs.uk/medicines/pantoprazole/how-and-when-to-take-pantoprazole/
91. How and when to take metoprolol - NHS, accessed November 12, 2025, https://www.nhs.uk/medicines/metoprolol/how-and-when-to-take-metoprolol/
92. Metoprolol: Package Insert / Prescribing Information / MOA - Drugs.com, accessed November 12, 2025, https://www.drugs.com/pro/metoprolol.html
93. Svelecte - autocomplete component in svelte - Vercel, accessed November 12,

2025, https://svelecte.vercel.app/searching
94. Svelecte - select with autocomplete written in Svelte, accessed November 12, 2025, https://mskocik.github.io/svelecte/
95. Simple Autocomplete / typeahead component for Svelte - GitHub, accessed November 12, 2025, https://github.com/pstanoev/simple-svelte-autocomplete
96. jspdf vs pdfmake vs html2pdf.js vs html-pdf | PDF Generation Libraries Comparison, accessed November 12, 2025, https://npm-compare.com/html-pdf,html2pdf.js,jspdf,pdfmake
97. A full comparison of 6 JS libraries for generating PDFs - DEV Community, accessed November 12, 2025, https://dev.to/handdot/generate-a-pdf-in-js-summary-and-comparison-of-libraries-3k0p
98. Creating PDFs from HTML + CSS in JavaScript: What actually works - Joyfill, accessed November 12, 2025, https://joyfill.io/blog/creating-pdfs-from-html-css-in-javascript-what-actually-works
99. Compare Top Node.js HTML to PDF Libraries - Open-Source and Commercial - DocRaptor, accessed November 12, 2025, https://docraptor.com/node-html-to-pdf
100. Generating PDFs from HTML with jsPDF and javascript - Pdforge, accessed November 12, 2025, https://pdforge.com/blog/generating-pdfs-from-html-with-jspdf
101. SvelteKit | Tauri v1, accessed November 12, 2025, https://tauri.app/v1/guides/getting-started/setup/sveltekit/
102. Pantoprazole: MedlinePlus Drug Information, accessed November 12, 2025, https://medlineplus.gov/druginfo/meds/a601246.html
103. Remeron Pill Images - What does Remeron look like? - Drugs.com, accessed November 12, 2025, https://www.drugs.com/image/remeron-images.html