**Developer's Guide 🛠️**

This guide provides technical information for maintaining and extending the PayTrax application.

**File Structure & Core Concepts**

The application is built with a modular vanilla JavaScript architecture. Understanding the role of each file is key to making changes.

* **PayTrax.html**: The single entry point and container for the application's UI.
* **style.css**: Contains all styling for the application.
* **/js/main.js**: The orchestrator. It handles event listeners (user actions) and calls functions from the other modules to update the state and UI.
* **/js/state.js**: The single source of truth. It defines the appData object structure, contains default values, and manages saving/loading data to IndexedDB or localStorage. All data modifications should happen by changing the appData object.
* **/js/logic.js**: The "brains" of the application. Contains all business logic: payroll calculations, tax computations, report generation, and date logic. These functions read from appData and return calculated values or modify appData directly.
* **/js/ui.js**: The "hands" of the application. Contains all functions that manipulate the DOM. These functions are responsible for taking data from the appData object and rendering it to the screen (e.g., populating dropdowns, updating tables, showing modals).
* **/js/data-io.js**: Handles the import and export of the appData object to and from JSON files.
* **/js/db.js**: A low-level module for interacting with the IndexedDB API.
* **/js/migration.js**: Contains the logic for upgrading the appData object structure from older versions to the current version during a data import.

**State Management**

The application is state-driven. The entire application's state is held within the appData object defined in state.js.

* **Rule 1**: To change what's on the screen, you must first change the data in the appData object.
* **Rule 2**: After changing the appData object, call the relevant function from ui.js to re-render that part of the screen.
* **Rule 3**: Call saveData() from state.js after any significant state change to persist it.

**Data Versioning & Migration**

To ensure backward compatibility with older data backups, the application uses a manual, developer-managed versioning system. When a change is made that alters the structure of the appData object, the developer must perform the following steps:

1. **Increment Version Constant**: In js/state.js, increment the CURRENT\_VERSION constant.  **Add New Properties to Default State**: In js/state.js, add the new data property to the defaultAppData object so that new users get the updated structure. . For example:

// js/state.js

export const CURRENT\_VERSION = 3; // Was previously 2

1. **Add New Properties to Default State**: In js/state.js, add the new data property to the defaultAppData object so that new users get the updated structure.
2. **Write Migration Script**: In js/migration.js, write a new function named migrateToV\_X\_ (where X is the new version number). This function takes the data object as an argument and must perform the necessary transformations. This could include adding new properties with default values, renaming properties, or restructuring parts of the object. **Crucially, the function must end by setting data.version = X;**.

// js/migration.js

function migrateToV3(data) {

// Add a new "notes" field to each employee

data.employees.forEach(emp => {

if (emp.notes === undefined) {

emp.notes = "";

}

});

data.version = 3; // Stamp the data with its new version

}

1. **Update the Migration Path**: In js/migration.js, add a new case to the switch statement inside the migrateData function. The cases are designed to fall through, ensuring that a very old backup can be brought up-to-date through a series of sequential migrations.

// js/migration.js

export function migrateData(data) {

const importVersion = data.version || 1;

switch (importVersion) {

case 1:

migrateToV2(data);

case 2:

migrateToV3(data); // Add the new case here

}

return data;

}