Lexica Draft Pro - Detailed Technical Documentation

Version: 1.0 (Stable Multi-Party Build) **Date:** September 7, 2025

Table of Contents

1. Introduction & Core Philosophy

1.1. Project Goal

Lexica Draft Pro is a client-side, browser-based application for creating complex, modular legal contracts. Its primary goal is to provide maximum flexibility to the end-user, allowing them to assemble documents from a rich library of pre-defined clauses, add their own custom clauses, and reorder content freely. All user data is stored in the browser's `localStorage` to ensure privacy and session persistence without requiring a server backend.

1.2. Architectural Choice: Single-File Application

The entire application is contained within a single `.html` file. This was a deliberate choice for several reasons:

- **Portability:** The entire application can be shared and run easily as a single file.
- **Simplicity:** Eliminates the need for build tools, bundlers (like Webpack), or complex server setups for this phase of the project.
- **Self-Contained:** All code, styles, and data are co-located, making it straightforward to understand the complete system at a glance.

2. File Structure & Technologies

2.1. Core Technologies

- **HTML:** Semantic HTML5 for the application's structure.
- **Tailwind CSS:** All styling is handled via utility classes, loaded from a CDN. Custom styles are minimal and contained in a `<style>` block.
- **JavaScript (ES6 Modules):** A single `<script type="module">` contains all application logic.
- **Feather Icons:** For UI iconography.
- **SortableJS:** Enables drag-and-drop functionality.
- **html-docx-js & FileSaver.js:** Used for the "Download as .docx" feature.

2.2. In-File Structure

The `lexica_draft_pro_final_stable.html` file is organized as follows:

1. **`<head>`:** Contains meta tags, CDN links for all external libraries (CSS, JS), and a `<script>` block for Tailwind CSS configuration. The primary custom styles for components like clause blocks are also defined here.
2. **`<body>`:**
    - **`<div id="app">`**: The main container for the entire application.
    - **`<header>`**: The persistent top navigation bar.
    - **`<main>`**: Contains the "pages" of the application. Each page is a `<div>` with a unique ID (e.g., `<div id="dashboard-page">`). Only one is visible at a time.
    - **Modals:** The AI Assistant modal is defined at the bottom of the body.
3. **`<script type="module">`**: The application's "brain." It contains all the JavaScript code, which is executed after the HTML body is parsed.

3. Core Architecture: A Deep Dive

The application operates on a state-driven rendering model. The UI is a direct reflection of the central `state` object. When the state changes, dedicated functions re-render the necessary parts of the UI.

3.1. The Data Model: `clausesDB` and `templatesDB`

These two constant objects are the heart of the application's content.

- **`clausesDB`**:
    - **Structure:** An object of objects, where each key is a camelCase `clauseId`.
    - **`text: (data) => \...`**: The most critical property. It's a **function**, not a static string. This allows each clause to be dynamic. When rendered, it's passed the current draft's formData object, enabling it to insert party names, dates, and other details directly into the clause text using template literals (${data.partyName}). The function for partiesRecitals is the most complex, as it dynamically builds an HTML table based on the number of parties.
- **`templatesDB`**:
    - **Structure:** An object of objects, where each key is a camelCase `templateId`.
    - **`defaultClauses`**: This array of `clauseId`s is the "recipe" for a contract. When a user selects a template, this array is used to populate the initial set of selected clauses in the wizard.

3.2. The State Machine: The `state` Object

This single object tracks everything the user is doing.

- `currentPage`: Managed by the `showPage()` function to control which `<div>` is visible.
- `currentDraftId`: Set when a user creates a new draft or opens an existing one. It's the key used to find the draft object within the `state.drafts` array.
- `formData`: A temporary holding place for the data being entered in the wizard. When the user clicks "Generate Draft," this data is copied into the new draft object.

    - `formData.parties`: This is an array, making the multi-party system possible. Each element is an object `{ name, role, address }`.
- `activeClauseIds`: This array holds the `clauseId`s for the draft currently being edited. In the wizard, it's updated every time a checkbox is clicked. In the editor, it is derived from the DOM when saving.
- `drafts`: The "master list" of all contracts. Each object in this array represents a saved contract and contains its `id`, `title`, `formData`, `activeClauseIds`, and `clausesContent` (the saved HTML of edited clauses). This entire array is what gets `JSON.stringify`'d and saved to `localStorage`.
- `sortableInstance`: Holds the active SortableJS object so it can be destroyed when leaving the editor page, preventing memory leaks.

3.3. The Rendering Engine

There is no complex framework like React. Instead, rendering is done via a set of plain JavaScript functions that build HTML strings and set the `innerHTML` of target elements.

- **Example Flow:** When `renderEditor()` is called, it finds the current draft in `state.drafts`. It then iterates through that draft's `activeClauseIds` array. For each ID, it calls `generateClauseHTML()`, which in turn retrieves the clause object from `clausesDB` and executes its `text(data)` function to get the dynamic content. All these HTML strings are joined together and injected into the editor container.

3.4. Execution Flow & Initialization

1. **DOM Load:** The `DOMContentLoaded` event fires.
2. **`init()`:** This is the starting gun.
3. **`loadState()`:** The app first checks `localStorage` for any saved `drafts` or `userProfile` data and populates the `state` object. This makes the app persistent.
4. **`resetWizard()`:** Prepares the wizard form with default data (e.g., pre-populating Party A with profile data).
5. **`renderTemplateSelector()`:** Dynamically builds the template selection page from the `templatesDB`.
6. **`updateUIWithProfileData()`:** Populates the UI with the user's name, etc.
7. **`goBackToDashboard()`:** Renders the recent drafts from the now-populated `state.drafts` array and shows the dashboard page. The app is now idle and waiting for user input.

4. Feature Implementation: Granular Breakdown

4.1. The Dynamic Multi-Party System

- **UI:** `renderPartiesContainer()` is the core function. It maps over `state.formData.parties` and generates a block of HTML for each party. Crucially, it only adds a "Remove" button if `index > 1`, ensuring there are always at least two parties.
- **State Management:** The "Add Party" button simply does `state.formData.parties.push(...)` and calls `renderPartiesContainer()`. The "Remove" button uses `state.formData.parties.splice(index, 1)` and re-renders.
- **Data Collection:** When "Generate Draft" is clicked, the code iterates over the party form elements in the DOM and builds the final `parties` array to be saved in the draft object.

4.2. Clause Reordering: The SortableJS Integration

- **Initialization:** In `renderEditor()`, a new `Sortable` instance is created.
- **Key Configuration:**

    - `handle: '.drag-handle'`: This is essential. It tells SortableJS that a drag can *only* be initiated by the "move" icon (`<i data-feather="move"></i>`), not by clicking anywhere on the clause.
    - `filter: '.clause-inserter, [contenteditable="true"]'`: This prevents dragging from being initiated when a user tries to click on the inserter buttons or select text to edit.
- **Saving:** SortableJS modifies the DOM directly. The application doesn't need to track the reordering in real-time. When the user clicks "Save Draft," the code simply reads the `.clause-block` elements from the editor *in their new order* and saves that sequence of `clauseId`s to the `activeClauseIds` array in the draft object.

4.3. Custom Clause Insertion System

- **UI:** The `generateInserterHTML()` function creates the inserter button. `renderEditor()` is responsible for placing one of these buttons between every clause block.
- **Logic:** When an inserter button is clicked, `generateCustomClauseHTML()` is called. This function creates the HTML for a new, fully editable clause block with a unique, timestamp-based ID (e.g., `custom_166258...`). This new block is inserted directly into the DOM before the inserter that was clicked.

4.4. DOCX Export: Achieving Professional Formatting

This feature is complex because web (HTML/CSS) and Word document formatting are fundamentally different. The alignment issues were solved by moving away from CSS-based indentation to a more rigid, structural approach that Word understands perfectly.

1. **Cloning:** The process starts by cloning the editor content into a temporary `<div>` to avoid modifying the live view.
2. **Clause Iteration:** The code loops through each `.clause-block` in the clone. A `clauseCounter` is maintained.
3. **Title Formatting:** The `<h4>` title is replaced with a simple `<p><strong>` tag containing the number (e.g., **"1. Definitions"**). This is more reliable for DOCX conversion than styled headers.
4. **Hierarchical Paragraph Numbering (The Core Fix):**
    - For each clause, a `paraCounter` is reset to 1.
    - The code iterates through the child nodes of the `.clause-content` div.
    - For each `<p>` or `<li>` tag with text, it generates a **borderless HTML table**. This table has two cells:
        - A narrow left cell containing the number (e.g., **"1.1"**).
        - A wider right cell containing the paragraph's original content.
    - This table structure **forces** Word to maintain the hanging indent and alignment, as it's a structural layout, not a stylistic one.
5. **Conversion:** The final, transformed HTML (now full of simple tables) is passed to `html-docx-js` to be converted into a `Blob`, which is then served to the user for download via `FileSaver.js`.

5. Extending the Application: A "How-To" Guide

5.1. How to Add a New Clause

1. Navigate to the `clausesDB` object in the script.
2. Add a new key-value pair. The key must be a unique camelCase ID.

```
// BEFORE
const clausesDB = {
    // ... existing clauses
    sla: { /* ... */ }
};
```

```
// AFTER
const clausesDB = {
    // ... existing clauses
    sla: { /* ... */ },
    dataSecurity: {
        id: 'dataSecurity',
        category: 'Contract-Specific',
        title: 'Data Security Measures',
        text: (data) => `<p>The receiving party agrees to implement industry-
standard security measures to protect all Confidential Information received under
this agreement.</p>`
    }
};
```

5.2. How to Add a New Contract Template

1. Navigate to the `templatesDB` object.
2. Add a new key-value pair. The key must be a unique camelCase ID.
3. Populate the `defaultClauses` array with `clauseId`s from `clausesDB`.

```
// BEFORE
const templatesDB = {
    // ... existing templates
    loanGuarantee: { /* ... */ }
};

// AFTER
const templatesDB = {
    // ... existing templates
    loanGuarantee: { /* ... */ },
    dataProcessingAgreement: {
        category: "IP & IT",
        title: "Data Processing Agreement (DPA)",
        description: "Standard terms for GDPR and data processing compliance.",
        defaultClauses: ['partiesRecitals', 'definitions', 'confidentiality',
'dataSecurity', 'returnOfInfo', 'governingLaw']
    }
};
```

The application will automatically detect and display these new additions in the UI.

6. Conclusion & Future Roadmap

This application provides a powerful, client-side solution for contract drafting. While feature-complete for a single-user experience, the clear next step (as explored in previous design phases) is to move towards a collaborative, multi-user platform.

This would involve:

- **Replacing `localStorage` with a real-time database** (e.g., Firebase Firestore).
- **Implementing user authentication** instead of the current anonymous profile system.
- **Building features for real-time multi-user editing, commenting, and role-based access control.**

The current component-based functions and state management provide a solid foundation for such a migration.