Design Rationale

Giacomo Bonomi

General About-

Thought of this project when I was doing a leetcode question and had to change three variable names due to a misspelling error. Many spelling/typing test websites online lack variability and have weird catches that make the website unusable (no deleting, weird texts, etc.)

Generative AI Disclaimer:
Most simple HTML/CSS elements, namely layout, design, and coloring, are GPT-Generated.

High-Level Abstraction:

IndexeDB - Storing the history of previous results between runs
History Manager Class - Handles history storage and info retrieval of runs.
Typing Test Class - Manages typing test logic and statistics calculation

Design within classes:

historyManager.js

This class essentially handles everything that will be stored in IndexeDB. It interacts directly with the DOM to display the results using a modal element.

Justification - Handling all history elements in one place greatly reduces overhead and prevents me from dealing with IndexeDB elements elsewhere. Helper functions directly interact with markdown elements, and the code was refactored to utilize asynchronous calls.

IndexeDB also allows me to not worry about user authentication and data synchronization, it's concurrency support also allows me to expand for future database applications.

Limitations and Considerations -
Miixing data retrieval and UI updates in one class is less than ideal. In a bigger system, I might separate these concerns more rigorously or integrate a front-end framework that would handle reactivity on its own.

testCore.js

Justification - The typing test logic—timing, accuracy, WPM calculation—exists here so it can be reused or expanded more easily without always touching the database or DOM. I rely on straightforward JavaScript timing (e.g., setInterval) and do my math for WPM based on how many words were typed within the elapsed time.

Limitations and Considerations -

If I wanted concurrency or a more robust scheduling system (for multiple user requests), I'd probably have to do more advanced logic. (Not a concern yet, especially since js is single-threaded).

This class still references some DOM elements (like the typing box). A more decoupled approach might require passing in callbacks or using an event emitter pattern.

main.js

Justification -I wanted one place to initialize everything once the DOM is loaded. This script ties together the HistoryManager and TypingTest, along with button click handlers, modals, and so on. When the test finishes, it feeds the resulting stats back into the HistoryManager to store them.

Limitations and Considerations -
Overriding finishTest is at best a temporary fix to inject my storage logic without rewriting the entire TypingTest class. A rewrite is definitely needed. probably with a dedicated state management library.

General Design Choices:

Centralized IndexedDB Handling

All history operations (initializing, reading, writing, clearing) are encapsulated within a single class, reducing redundant IndexedDB code and ensuring a uniform interface.

Direct DOM Manipulation via Helper Functions

Helper functions are used to update markup elements directly based on history data, enabling consistent rendering of history entries and modals without scattering DOM queries throughout the code.

Use of Asynchronous Calls

Asynchronous operations (using Promises and async/await) are integrated to handle non-critical background tasks such as data fetching and updating. This ensures that UI updates and essential interactions remain responsive while waiting for IndexedDB transactions to complete.