Student Points MVP — UI + Spreadsheet Integration

Conversation summary and next steps (generated 2026-01-20)

1. Project goal (MVP scope)

Build a teacher-facing web page to add/deduct student points across four grades. This MVP intentionally starts with **no login** and **no persistence** so the core UI and workflow can be validated quickly. Later iterations connect the data to a spreadsheet as a lightweight database.

2. What was built first

- A consistent UI style system (CSS) applied to all components.
- Static layout (HTML) with a two-panel workspace: left navigation + right operations panel.
- Working point operations (JavaScript) using two in-memory data structures: **students** and **events**.

3. Architecture explained in three layers

To make the system explainable (and defendable in an engineering notebook), the implementation was organized and documented using a **three-layer** mental model: **Data layer**, **Structure layer**, and **Rendering layer**.

Layer	Responsibility	Key artifacts (examples)
Data layer	Source of truth (state + audit trail)	students[], events[]; rebuildPointsFromEvents()
Structure layer	Static UI skeleton	tabs, search box, student list container; right-panel containers
Rendering layer	Bridge: state/data → DOM	renderAll(), renderTabs(), renderStudentList(), renderSelecte

4. Data model decisions (students + event log)

Two core datasets were used from day one. This makes the later spreadsheet/database step mostly an adapter swap rather than a rewrite.

- **students[]**: snapshot for UI listing (id, grade, class, name, points).
- **events[]**: append-only point event log (event_id, time, student_id, delta, type, description, operator).
- **Consistency strategy**: points can be rebuilt from the event log to avoid drift when events are corrected.

5. Rendering model (why renderAll exists)

A central render controller (**renderAll()**) calls the three render modules after each user action. This "re-render after every action" design is a deliberate MVP trade-off: simpler reasoning and fewer UI edge cases at the cost of extra work per action.

Universal UI pipeline:

User action → state/data update → **renderAll()** → UI refresh

6. Engineering notebook guidance captured in the chat

Key takeaway: the notebook should explain **system behavior and design choices** (module responsibilities, data flow, trade-offs), not every line of JavaScript. Helpful conventions include inline code formatting for identifiers (e.g., **`events`**), short code blocks for 1–2 critical functions, and pseudo-code for workflows when syntax-level detail is not the focus.

7. Spreadsheet integration plan (what “connect to Sheets” really means)

The goal was to replace in-memory arrays with a storage adapter backed by Google Sheets. Because a static HTML page cannot safely write to Sheets directly, a minimal middle layer is needed: **Google Apps Script deployed as a Web App** (acts like an API).

Recommended sheet layout

Tab	Role	Suggested columns
Students	Roster (mostly static)	id | grade | class | name (points optional; can be derived)
Events	Append-only audit log	event_id | time | student_id | delta | type | description | operator

API surface (minimal functions)

- **GET /?action=students** → load roster
- **GET /?action=events** → load event log
- **POST** {action:"appendEvent", event:...} → append a new row to Events
- **POST** {action:"updateEvent", event_id, patch} → locate a row by event_id and update fields

8. Debugging incident and resolution

Symptom: opening the Web App URL in a browser returned correct JSON, but the page UI remained empty. After adding defensive initialization (try/catch + toast), the error surfaced as:

```
TypeError: Failed to fetch
```

Resolution: the front-end needed to be served via HTTP (not opened as a local file). Running a local server fixed the fetch restrictions and allowed the page to call the Apps Script endpoint.

Command used:

```
python3 -m http.server 8000
```

9. Next steps (action plan)

- **Upload to GitHub**: clean repo structure (main.html, README, docs/ screenshots/). Add a short README with setup steps (local server + Apps Script URL).
- **Write a developing log entry**: summarize the iteration (UI MVP → layer documentation → spreadsheet adapter → fetch bug → fix) and record key design trade-offs.
- **Start the next system**: begin scaffolding the second module (define requirements, data model, minimal UI skeleton, then connect to storage only after workflow is proven).