**Implementation and Development Plan**

This plan outlines the implementation details and development roadmap for a **local-only -style gamified classroom web app**. The application will run fully offline on a Windows machine, using local JSON files for storage and Node.js for all server-side logic. It is designed to work in both mobile and desktop browsers, using GitHub for version control and continuous integration. The plan is structured into sections covering the technology stack, development steps, architecture, data handling, testing, deployment, and CI practices.

**1. Recommended Tech Stack**

**Platform & Language:** The app will be built with **Node.js** on the backend and a single-page application on the frontend. Node.js is cross-platform (works on Windows) and provides the built-in fs module for file I/O, which we will use to read/write JSON files for persistence ([Reading and writing JSON files in Node.js: A complete tutorial - LogRocket Blog](https://blog.logrocket.com/reading-writing-json-files-node-js-complete-tutorial/#:~:text=Node.js%20provides%20built,specifically%20for%20working%20with%20files)). All game logic (XP, HP, AP calculations, etc.) will run in Node modules, ensuring consistency in one place (Development Guidelines for JSON-Based -Style Web App.docx).

**Backend Framework:** We will use **Express.js** (a minimalist web framework for Node) to create a local REST API. Express allows defining HTTP endpoints that modify or retrieve data, which is suitable for an offline local server. According to the specifications, the backend should expose a REST API to read/write the JSON “database” safely (Local Gamified Classroom Web App.docx). Express is a lightweight choice that runs locally with no internet connectivity needed. It also easily serves the frontend files and supports mobile clients connecting over a local network. In an offline setup, all API calls will simply affect local JSON data (Development Guidelines for JSON-Based -Style Web App.docx) (Development Guidelines for JSON-Based -Style Web App.docx), with no external calls.

**Frontend Framework:** The frontend will be a **responsive SPA (Single Page Application)** built with **React** (as recommended) (Development Guidelines for JSON-Based -Style Web App.docx). React is chosen for its component-based architecture and strong community support. It can render dynamic dashboards and update the UI in real-time as game state changes. React apps run in any modern browser, and we will ensure mobile compatibility via responsive design (using CSS flexbox/grid or a UI library like Bootstrap/Tailwind for layout). This means students on tablets or phones can use the app through their browser, with the interface adapting to smaller screens. Other SPA frameworks (Vue, Angular) could work, but React is preferred given the guidelines (Development Guidelines for JSON-Based -Style Web App.docx) and its ease of integration with a Node/Express backend.

**Data Storage:** All persistent data will be stored in local **JSON files** (flat text files) under a /data directory. This takes the place of a database. For example, we will have files like users.json, characters.json, teams.json, powers.json, quests.json, etc., as outlined in the project spec (Local Gamified Classroom Web App.docx). Node’s fs module will handle reading and writing these JSON files. Using JSON makes the app completely offline-capable and easy to back up or inspect, since all data is human-readable. It also aligns with the **Do** directives of the project to use local .json files instead of any cloud or SQL database (Development Guidelines for JSON-Based -Style Web App.docx). The schema of these JSON files will mimic the relational structure of game data, but stored as nested JSON objects for simplicity (e.g. an array of character objects in characters.json).

**Mobile and Desktop Compatibility:** The combination of a React front-end and an Express/Node back-end supports both desktop and mobile browsers. We will ensure the UI is touch-friendly and tested on mobile browsers. The app being a web app means it’s inherently cross-platform. Additionally, we might leverage Progressive Web App (PWA) techniques for the front-end so that assets can be cached and load quickly on repeat visits (though an active connection to the local server is needed for data sync). The use of standard HTML5/CSS3 and responsive design will address different device sizes.

**Additional Libraries/Tools:** We will use some supporting libraries to streamline development:

* **File system helpers:** Possibly use fs-extra or Node’s fs.promises for convenient JSON read/write. We may also use a library for atomic file writes to prevent corruption (more on file safety later).
* **Front-end UI libraries:** We can utilize a component library (like Material-UI or Bootstrap) for faster UI development, ensuring consistent styling and mobile-ready components (buttons, modals, etc.).
* **State management:** If needed, a client-side state management library (like Redux or Zustand) can manage global state (though for an MVP, React’s built-in state and context might suffice).
* **Build tools:** Use a bundler or scaffolding tool like Create React App or Vite for the React project. This provides a development server for hot reloading and a production build process for optimizing assets. The final build will be served by our Node server.
* **Version Control:** All code will be managed with **Git**, and hosted on **GitHub** for collaboration. We will initialize a Git repository at project start, commit frequently, and use branches for feature development. This enables code reviews via pull requests and easy rollback if needed. Using GitHub also sets us up to integrate Continuous Integration (CI) workflows later for automated tests on each commit.

**Optional – Electron:** For a desktop deployment option, we can use **Electron** to wrap the Node backend and React frontend into a desktop application. Electron would allow bundling the app into an executable that runs a Chromium browser with Node integration. This could provide a native app feel on the teacher’s Windows PC and even operate without a separate browser. In Electron, the front-end could communicate with the back-end via IPC (inter-process communication) instead of HTTP, but the core logic and data storage would remain the same (Development Guidelines for JSON-Based -Style Web App.docx). We consider Electron optional; it’s useful if a teacher wants to run everything in a single desktop app or if packaging the app for easy distribution, but it’s not required for normal browser-based use.

**Summary:** This tech stack (Node/Express + React + JSON files) meets all requirements: it is fully offline, runs on Windows, works in browsers on any device, and uses no external database or internet services. The tools chosen are popular and well-supported, making development and maintenance easier.

*Table 1: Key Technologies and Rationale*

| **Aspect** | **Technology / Tool** | **Rationale** |
| --- | --- | --- |
| **Back-End Runtime** | Node.js (JavaScript) | Cross-platform (Windows OK), access to fs for file I/O ([Reading and writing JSON files in Node.js: A complete tutorial - LogRocket Blog](https://blog.logrocket.com/reading-writing-json-files-node-js-complete-tutorial/#:~:text=Node.js%20provides%20built,specifically%20for%20working%20with%20files)), aligns with JavaScript skillset. |
| **Server Framework** | Express.js (Node library) | Simple REST API creation, lightweight, works offline, can serve static files (frontend). |
| **Front-End Framework** | React (SPA) | Recommended by spec (Development Guidelines for JSON-Based -Style Web App.docx); modular UI components, fast DOM updates, good for dynamic dashboards; wide community support. |
| **Data Storage** | Local JSON files (via Node fs) | No DB server needed (fully offline), human-readable, easy to version control initial data, meets project requirements (Local Gamified Classroom Web App.docx). |
| **Styling/UI** | CSS3 + Responsive framework | Ensure mobile-friendly design; possibly use Bootstrap/Tailwind for grid system and quick styling. |
| **Testing Framework** | Jest (for JS) + React Testing Library | Enables unit tests for game logic and React components; widely used and integrates with CI. |
| **Build & Tooling** | Create React App or Vite; Webpack | Bundle frontend for production; dev server for live reload. Also use ESLint & Prettier for code quality. |
| **Desktop Packaging** | *Optional:* Electron | Allows bundling app into a desktop executable for one-click launch; not needed for browser use but available for convenience. |
| **Version Control** | Git (GitHub) | Track changes, enable collaboration. Required for CI integration and GitHub hosting of code. |

**2. Step-by-Step Development Roadmap**

We will approach development in phases, from initial setup to MVP completion and then polishing for a production-ready release. Below is a step-by-step roadmap:

**Step 1: Project Setup** – Begin by setting up the development environment and project structure. Initialize a Node.js project (npm init) and a Git repository. Set up two main folders: backend/ for server logic and frontend/ for the React app. Create a basic Express server in server.js that serves a "Hello World" and perhaps a placeholder API endpoint. Similarly, scaffold the React app (using Create React App or a minimal Vite template) in the frontend folder. Verify that the frontend can be built and that Express can serve the production build. Commit this initial scaffold to GitHub. This establishes the foundation to build upon and ensures that the tools (Node, npm, build process) are correctly configured on the Windows machine.

**Step 2: Define Data Schema & JSON Files** – Plan out the JSON structure for game data and create the initial files in the /data directory. Following the spec, create files like users.json, characters.json, teams.json, classes.json, powers.json, quests.json, etc. with some sample data or empty arrays/objects (Local Gamified Classroom Web App.docx). For example, define a few sample classes (Warrior, Healer, Mage) in classes.json with their base stats and powers, and a couple of sample users/characters for testing. This schema design is important to get right early since game logic will depend on it. Ensure each entry has unique IDs and that relationships are represented by IDs (e.g. a character object stores userId, teamId, etc.). Commit the schema and sample data.

**Step 3: Implement Core Game Logic (Backend Modules)** – Develop the backend game logic layer as modular JavaScript files under /backend/. Each module will handle a specific aspect of the game rules, as outlined in the design document (Development Guidelines for JSON-Based -Style Web App.docx):

* xpSystem.js: Functions to award XP to a character or deduct XP (if negative behavior). This should update the character’s XP in characters.json, append an entry to an XP log (e.g. game\_actions.json or xp\_log.json), and check for level-ups if using an XP-to-level system.
* hpSystem.js: Functions to deduct health points from a character. This updates HP in characters.json and checks for "fall in battle" (if HP drops to 0 or below). If a character falls, possibly trigger a team consequence (like their teammates lose some HP too) as per rules.
* apSystem.js: Manages Action Points. Functions to reduce AP when a power is used, and possibly regenerate AP (e.g., restore AP daily or on certain events). Ensure AP never goes below 0 and perhaps has a max cap.
* powerSystem.js: Validates and executes character powers. Each power (defined in powers.json) has prerequisites and effects. This module will check if a character has enough AP to use a power and apply the power’s effects. For example, a “Protect” power might transfer damage from a teammate to the caster – the logic for that would live here (Development Guidelines for JSON-Based -Style Web App.docx).
* teamSystem.js: Handles team-wide effects. For instance, if one team member falls in battle, all team members might lose HP (shared consequence) (Minimum Viable Product (MVP) Definition.docx). Or if XP is awarded to a team (group reward), distribute it. This module works in conjunction with xpSystem and hpSystem when those events have team implications.
* eventSystem.js: Manages random events or daily events triggered by the teacher. For MVP, this could be simple (e.g., a function to apply a random effect from events.json to some students). This is less critical than the core XP/HP/AP but is listed for completeness.

Each of these modules will load and update the JSON data as needed. At this stage, implement these as pure functions that take in relevant IDs or data and then read/write to the JSON files (using Node fs). Initially, use synchronous file operations for simplicity (e.g., fs.readFileSync and fs.writeFileSync) to ensure data is updated immediately (Development Guidelines for JSON-Based -Style Web App.docx). We will refine file access later for safety. Write basic unit tests for these functions (e.g., test that adding XP increases a character’s XP and that HP cannot drop below 0, etc.) to validate the logic in isolation. This step completes the core game logic layer before building any UI.

**Step 4: Develop API Endpoints (Local REST API)** – Create an Express router (or a set of route handlers) that maps HTTP API calls to the backend logic functions. For example, implement routes such as:

* POST /api/characters/:id/xp – to award XP to the character with given id (calls xpSystem.addXp(characterId, amount) then returns updated character data) (Development Guidelines for JSON-Based -Style Web App.docx).
* POST /api/characters/:id/hp – to deduct HP (calls hpSystem.deductHp(characterId, amount)).
* POST /api/characters/:id/power – to use a class power (calls powerSystem.usePower(characterId, powerId) and applies effects).
* GET /api/characters/:id/status – to retrieve a character’s full status (XP, HP, AP, level, powers, etc.) (Development Guidelines for JSON-Based -Style Web App.docx).
* GET /api/team/:id – to fetch team data (members, shared stats).
* GET /api/quest/:id – to fetch quest details or progress.
* POST /api/quest/:id/advance – to advance a quest step (e.g., marking a step complete and awarding any XP rewards) (Development Guidelines for JSON-Based -Style Web App.docx).

These correspond to the basic routes outlined in the guidelines (Development Guidelines for JSON-Based -Style Web App.docx). Each route will use Express’s JSON body parsing for POST data (e.g., the XP amount to add might come in the request body). The route handler will perform any necessary validation (e.g., check that the user is a teacher for awarding XP, or that the power being used belongs to that character) and then call the appropriate backend module function. The function will update the JSON file and the API will respond with the new state (or a success message). Since this app runs locally and offline, we don’t need extensive authentication tokens; however, we will simulate a simple auth by checking user roles (perhaps storing a “current user role” in memory or using a cookie to differentiate teacher vs student for certain endpoints). At this stage, test the API manually using tools like Postman or cURL on localhost to ensure each endpoint modifies the JSON as expected. This provides a backend that is ready to be consumed by the front-end.

**Step 5: Build Frontend MVP Interfaces** – With the backend in place, focus on the React front-end. Start by creating the essential views and components:

* **Login Screen:** A simple login page where a user selects whether they are a teacher or a student and possibly picks their name/character (for MVP, we might skip actual credential verification and allow selecting from a list of users for convenience).
* **Student Dashboard:** After “login”, a student sees their character’s dashboard. Build the DashboardView component to display the character’s current XP, HP, AP, level, gold, and list of available powers (Local Gamified Classroom Web App.docx) (Local Gamified Classroom Web App.docx). Include interactive elements like a button to use each power (which will trigger a POST to the power API) and maybe a list of recent events or rewards.
* **Teacher Dashboard:** If the logged-in user is a teacher, show an overview of the class. The TeacherView component should list all students (characters) with their stats so the teacher can monitor everyone (Local Gamified Classroom Web App.docx). Include controls for the teacher to award XP or deduct HP (e.g., buttons or forms next to each student to trigger those actions, which call the respective API endpoints). Also provide a way to create a quest or trigger an event. Initially, for MVP, these could be very simple forms: e.g., a form to enter XP and select a student to award, or a dropdown of predefined events to trigger.
* **Team View:** A component (maybe accessible from the student dashboard) to show the student’s team members and their stats (Development Guidelines for JSON-Based -Style Web App.docx). This uses data from the /api/team/:id endpoint to list all characters on the team and could highlight if any teammate has fallen or needs help (encouraging collaboration).
* **Quest View:** A page for students to view a quest storyline assigned by the teacher (Development Guidelines for JSON-Based -Style Web App.docx). For MVP, a "quest" can be a simple list of steps or choices. The QuestView will display the current quest description and have a button to advance to the next step (if the quest is linear) calling /api/quest/:id/advance. When a quest is completed, it might award XP to the character.
* **Admin/Settings (optional for MVP):** A view for the teacher to configure game rules or edit quests. This could be as simple as allowing them to edit JSON via the UI, but for MVP we might limit scope and just allow quest creation in a basic way (entering text for steps and rewards).

While building these components, ensure the layout is responsive. Use a mobile-first approach: for example, the student dashboard might stack the stat bars and power buttons vertically on a narrow screen, but lay them out in a side-by-side grid on desktop. Use React state and effects to fetch data from the API endpoints and refresh the view. For example, when the DashboardView mounts, fetch the character status from /api/characters/:id/status to display current values. Also implement periodic polling or refreshing of data to mimic real-time updates (since we are not using web sockets) (Development Guidelines for JSON-Based -Style Web App.docx). A simple approach is to use setInterval inside a React effect to re-fetch the status every, say, 5 seconds, so that if the teacher awards XP from their screen, a student’s browser will update shortly after (Development Guidelines for JSON-Based -Style Web App.docx). This polling interval can be made configurable or optimized later.

**Step 6: Integrate Frontend with Backend** – Connect the React UI with the Express API. This involves wiring up all buttons and forms to make fetch calls to the local API endpoints. Use the browser Fetch API or a library like Axios to send requests. Because the app is served from the same origin (e.g., http://localhost:3000 for frontend and API), no special CORS handling is needed. For example:

* When the teacher clicks "Award 50 XP" for a student, the front-end will POST /api/characters/123/xp with { amount: 50 } in the body. On success, perhaps update that student’s displayed XP immediately (optimistically) or refresh the data via GET.
* When a student uses a power, the front-end calls POST /api/characters/123/power with { powerId: "protect1" }. The server will respond with the outcome (e.g., new HP of the target teammate), and the UI should reflect any changes (maybe showing a message "You protected your teammate!").
* On the quest view, an "Advance" button triggers a POST to advance the quest. The response can include updated quest state or XP gained, which the UI then displays.

At this stage, we will likely uncover issues or missing pieces in the backend logic and adjust accordingly. For instance, we may need to add a questSystem.js to handle quest progression rules, or adjust xpSystem to handle XP rewards from quests. We will implement those as needed so that the end-to-end functionality works. By the end of this step, we should have a functioning MVP where a teacher can log in and perform core actions (award XP, deduct HP, create a quest) and a student can log in to see their status, use powers, and go on quests. All of this should work offline with the state saved in JSON. This fulfills the fundamental game loop and classroom interactions envisioned in the MVP definition (Minimum Viable Product (MVP) Definition.docx).

**Step 7: Achieve MVP Milestone** – Ensure all **MVP features** as defined are implemented and working. The MVP should include the character system with at least three classes (Warrior, Healer, Mage) and their stats/powers, XP/HP/AP mechanics, ability to use powers, a basic quest system, team functionality, and teacher controls (Minimum Viable Product (MVP) Definition.docx) (Minimum Viable Product (MVP) Definition.docx). At this point:

* Characters can gain XP and level up, and lose HP (possibly triggering consequences) – fulfilling the core behavior management loop.
* AP and powers are enforced (can only use if AP sufficient, and effects apply correctly) (Minimum Viable Product (MVP) Definition.docx).
* Teachers can manage quests and apply rewards/penalties.
* Students can view their progress and collaborate as a team.
* All data modifications persist in JSON files, and the app can run entirely without internet (Minimum Viable Product (MVP) Definition.docx).

We will conduct thorough manual testing of each use case (teacher giving points, student using a power, quest completion, etc.) to verify the MVP works as intended. This is also a good time to engage a few test users (if available) to try the app in a controlled offline setting, ensuring it’s intuitive and stable. Any critical bugs or UX issues uncovered will be fixed. Once the MVP is solid, mark this milestone (and consider tagging a release in GitHub).

**Step 8: Testing and Quality Assurance** – With the MVP features in place, formalize the automated testing. Expand the unit test suite for all game logic modules (if not done already) to cover edge cases (e.g., awarding XP that causes a level-up, using a power with exactly equal AP, multiple team members falling scenario, etc.). Also add integration tests for the API endpoints. Using a tool like SuperTest (for Express), we can simulate HTTP calls to our API and assert that the JSON response or the data file changes are as expected. For example, a test can POST XP to a character and then GET that character’s status to ensure the XP increased appropriately and did not exceed a maximum. Ensure test isolation by using a separate test JSON directory or by mocking the fs module to not overwrite the real data during tests. We will also test the front-end with a combination of unit tests (for small pure functions or reducers) and possibly integration/UI tests. For React, we can use **React Testing Library** to render components and simulate user actions, verifying that the component state or DOM updates happen (e.g. clicking "use power" button causes the AP value on screen to decrease). However, front-end integration tests can be complex to run in CI; as an MVP, we might focus more on backend logic tests and do manual UI testing. Our goal is to have good coverage on game logic and critical API paths, since these are the most crucial to get right (the educational gameplay mechanics must be reliable).

**Step 9: Performance and File Safety Improvements** – Before final deployment, consider optimizations like caching frequently used data in memory to reduce file reads, especially if the JSON files grow. For example, we could load all JSON data into memory on server start and update the in-memory structures on each change, writing back to disk periodically or on every change. However, given the likely moderate size of a classroom data, simple on-demand reading and writing may suffice in terms of performance (file operations on small JSON files are usually milliseconds). The more pressing concern is avoiding data corruption or race conditions. Therefore, implement the file write **safety strategy** (detailed in section 5 below) now if not already: ensure that multiple nearly-simultaneous writes don’t interleave. If we used synchronous writes, we largely avoid race conditions since Node will execute them one at a time, but if using async writes, introduce a queue mechanism. This step might involve using a library like proper-lockfile for true file locking or simply an internal queue that awaits each write completion before starting the next. Testing these conditions (e.g., two clients awarding XP at the same time) is part of this phase. Also, ensure that any critical section of code (read-modify-write sequence) is protected. For example, awarding XP might involve reading the current XP, adding to it, then writing – we want to prevent another award from reading the old value in between. A simple lock around that operation (even a JavaScript Promise-based lock) can suffice.

**Step 10: Final Polishing and Production Readiness** – Clean up the UI/UX based on feedback. This could include better input validation (e.g., prevent negative XP awards via UI), nicer styling and maybe adding some graphics or icons to make the game feel fun. Ensure that the application loads quickly and works offline: verify that no external CDNs are used (bundle any needed libraries locally) and that the app doesn’t try to contact any cloud service. Update documentation: write a README or user guide for the teacher explaining how to run the app and use its features. Also document the JSON file format so that advanced users (teachers) could even hand-edit if necessary or troubleshoot data. At this stage, consider if any **optional enhancements** (from the spec’s Phase 2) are desirable before release. For example, real-time updates via WebSockets could be added if time permits, to push changes instantly to clients (though not strictly necessary offline) (Local Gamified Classroom Web App.docx). Or packaging with Electron so that the teacher can simply double-click an app instead of running Node manually. These can be deferred if time is short, but noting them is good for future work.

By following this roadmap step-by-step, we progress from a basic setup to a fully functional MVP, and onward to a polished final product. Each step builds on the previous, ensuring at each stage the app remains in a working state (this is important especially since we’ll be committing to Git; we want our main branch always relatively stable). This phased approach also makes it easier to test incrementally and involve stakeholders (like showing an early demo to teachers with partial functionality to gather feedback). Throughout development, we will use GitHub to track issues and tasks, potentially setting up a project board to manage the roadmap steps and any bugs or feature requests that come up.

**3. Directory Structure and Module Responsibilities**

Organizing the project clearly will help multiple developers collaborate and keep concerns separated. Below is a proposed directory structure for the project, with each folder’s purpose and key modules explained:

project-root/

├── backend/ # All backend (game logic and server) code

│ ├── xpSystem.js # Module for XP gain/loss logic ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,gain%2Floss))

│ ├── hpSystem.js # Module for HP deduction and "fall in battle" checks ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,gain%2Floss))

│ ├── apSystem.js # Module for AP usage and regeneration logic ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,battle%20checks))

│ ├── powerSystem.js # Module for validating & executing class powers ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,costs%20and%20regeneration))

│ ├── teamSystem.js # Module for team-based effects (sharing damage/bonuses) ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,team%20consequences%2Frewards))

│ ├── eventSystem.js # Module for random events and their outcomes ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,random%20events%20and%20their%20triggers))

│ └── questSystem.js # (Optional) Module for quest progression and rewards

├── server.js # Express server setup: defines API routes and serves frontend

├── data/ # "Database" JSON files (local storage) ([Local Gamified Classroom Web App.docx](file://file-51b4t5Jg7aTvJvnNBuuLcU#:~:text=,teacher%20user%20accounts))

│ ├── users.json # User accounts (students & teacher profiles)

│ ├── characters.json # Characters and their stats (XP, HP, AP, class, level, etc.)

│ ├── teams.json # Teams and members mapping

│ ├── classes.json # Definition of classes (class name, base stats, etc.)

│ ├── powers.json # Definition of powers (id, which class, AP cost, effect) ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=)) ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=))

│ ├── quests.json # Quest storylines and progress state

│ ├── events.json # Random events definitions

│ ├── store.json # (Optional) Items available for purchase with gold ([Local Gamified Classroom Web App.docx](file://file-51b4t5Jg7aTvJvnNBuuLcU#:~:text=,and%20memberships))

│ ├── inventory.json # (Optional) Items owned by each character (if using gold) ([Local Gamified Classroom Web App.docx](file://file-51b4t5Jg7aTvJvnNBuuLcU#:~:text=,and%20memberships))

│ └── game\_actions.json # Log of game actions (XP awarded, HP lost, etc.) ([Local Gamified Classroom Web App.docx](file://file-51b4t5Jg7aTvJvnNBuuLcU#:~:text=,storylines%20and%20progress%20tracking))

├── frontend/ # Frontend React application (SPA)

│ ├── public/ # Static assets and index.html (if using CRA/Vite)

│ ├── src/ # Source code for React app

│ │ ├── App.jsx # Main React component / router

│ │ ├── index.jsx # Entry point, renders App

│ │ ├── views/ # Page-level components for various screens:

│ │ │ ├── LoginView.jsx # Login/role selection screen

│ │ │ ├── DashboardView.jsx # Student dashboard (XP, HP, AP bars, powers) ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=))

│ │ │ ├── TeamView.jsx # Team status view ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,AP%20bars%2C%20powers%2C%20recent%20events))

│ │ │ ├── QuestView.jsx # Quest progression interface ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,allow%20class%20powers%20like%20%E2%80%9CProtect%E2%80%9D))

│ │ │ ├── TeacherView.jsx # Teacher dashboard (manage class, award points) ([Development Guidelines for JSON-Based -Style Web App.docx](file://file-VD2ZXxyEPYpz618CPXhSm8#:~:text=,display%20with%20steps%20and%20branching))

│ │ │ └── SettingsView.jsx # (Optional) Admin settings for rules/behaviors

│ │ ├── components/ # Reusable lower-level components (e.g., stat bar, power button)

│ │ └── services/ # Helper modules (e.g., API client code for fetch calls)

│ └── package.json # Frontend-specific dependencies (if managed separately)

├── tests/ # Automated tests

│ ├── backend/ # Tests for backend logic

│ │ ├── xpSystem.test.js # Unit tests for XP system (XP addition, level up scenarios, etc.)

│ │ ├── hpSystem.test.js # Unit tests for HP system (HP deduction, death consequences)

│ │ ├── powerSystem.test.js # Unit tests for powers (AP cost enforcement, effect correctness)

│ │ └── ...etc.

│ ├── server.test.js # Integration tests for API routes (Express handlers)

│ └── frontend/ # Tests for frontend (if any)

│ ├── DashboardView.test.jsx # Example: test that dashboard renders stats correctly

│ └── ...etc.

├── package.json # Project dependencies and scripts (for backend and overall project)

├── README.md # Documentation for the project (setup, usage, etc.)

└── .github/workflows/ # CI configuration (GitHub Actions workflows)

└── ci.yml # CI script for linting, testing, etc.

**Module Responsibilities:** Each backend module in /backend encapsulates the game rules for a specific domain:

* *xpSystem.js:* Provides functions like addXp(characterId, amount) and removeXp (if needed). It handles incrementing XP and possibly determining if a character levels up (if using a level system). If a level-up occurs, it might reset XP overflow and increase level, and maybe grant new powers. It will log the XP gain event to game\_actions.json.
* *hpSystem.js:* Functions like deductHp(characterId, amount) and maybe healHp. It ensures HP doesn’t drop below 0. If HP hits 0, mark the character as "down" ( concept of falling in battle). It could invoke teamSystem to apply any team-wide penalties (for instance, sometimes has all teammates lose HP when someone falls).
* *apSystem.js:* Might include useAP(characterId, cost) and regenAP(characterId). The regen could be called from an external scheduler (like once per day all characters get full AP, or maybe AP resets each day). For MVP, we could simplify AP to only change when using powers and maybe fully replenish when a quest is completed or daily by a manual teacher action.
* *powerSystem.js:* A crucial module that checks if a given power (by id) can be used by a character. It will reference powers.json for the power’s details (which class, what it does) and ensure the character is of the correct class and level to have that power. It deducts the appropriate AP via apSystem and then executes the effect. Effects might include: protecting a teammate (so redirect damage via hpSystem), healing someone (increase HP via hpSystem within limits), or giving AP to another (for a Mage perhaps). Each power effect might touch multiple systems.
* *teamSystem.js:* Contains logic for shared rewards or penalties. For example, implement onCharacterDown(characterId) that finds that character’s team and applies HP loss to each teammate (simulating the mechanic where if a team member falls, others lose HP) (Minimum Viable Product (MVP) Definition.docx). It could also have rewardTeam(teamId, xpAmount) to give every member XP (for team quest completion, etc.).
* *eventSystem.js:* Has functions to trigger random events. Likely uses events.json which lists possible events (maybe an id, description, and an effect like "everyone loses 5 HP" or "bonus 10 gold to all"). The teacher might call triggerEvent(eventId) and this module will carry it out by updating relevant JSON (HP, gold, etc., depending on the event definition).

All these modules will interact with the data files. To avoid duplicating file I/O code, we might also create a simple data access layer (e.g., a dataService.js that has functions to get or save a particular JSON file). For instance, dataService.getCharacters() reads characters.json and parses JSON, returning an array of character objects; dataService.saveCharacters(charactersArray) stringifies and writes it. This helps keep file handling code in one place, making it easier to implement locking or changes in storage approach later. The game logic modules can then call these data functions to retrieve or update data.

**Frontend Structure:** The React app is divided into **views** (pages) and **components** (reusable pieces). The views correspond to routes or major sections of the app (login, student dashboard, teacher dashboard, etc. as listed above). Each view may compose multiple smaller components. For example, the DashboardView might use a StatBar component to render an XP bar and HP bar, and a PowerButton component for each power. This structure follows typical React project conventions and the guidance to include DashboardView, TeamView, QuestView, TeacherView, etc. (Development Guidelines for JSON-Based -Style Web App.docx). Keeping these in a views/ folder makes it clear they are page-level. Reusable components (like a modal dialog or a form input) can live in components/. We will also include any necessary context providers (for global app state, like current user or theme) in the src directory.

**Data Directory:** The /data folder holds all JSON files representing the application state. Each file corresponds to a domain of data:

* users.json: contains user accounts (at least one teacher and several students). Each user could have fields like id, name, role (teacher/student), and perhaps login credentials if we later secure the login.
* characters.json: an array of character objects. Each character links to a user (userId) and class, and stores stats like xp, hp, ap, level, power list, teamId, gold, etc. (Development Guidelines for JSON-Based -Style Web App.docx) (Development Guidelines for JSON-Based -Style Web App.docx). In this design, each student has one character (the teacher might not have a character or has a special admin character).
* teams.json: defines teams, which could be an array where each team has an id, a name, and a list of member character IDs. This makes it easy to lookup a character’s teammates by finding their team and listing all members.
* classes.json: defines base class information (Warrior, Healer, Mage as per MVP, and potentially others if expanded). Each class entry might include default HP, default AP, a description, and what powers are available at what levels.
* powers.json: defines each power/ability available in the game (Development Guidelines for JSON-Based -Style Web App.docx) (Development Guidelines for JSON-Based -Style Web App.docx). Fields might include id, name, class (which class can use it), tier or level requirement, cost (AP cost), and a description or effect definition.
* quests.json: holds quest data. A quest could be represented as a series of steps or checkpoints. For simplicity, each quest might have an id, title, steps (array of text or actions), and possibly a mapping of character progress (which step each character/team is on). The teacher’s quest creation would add entries here.
* events.json: lists random events, each with an id, description, and effect. Effects could be encoded in a simple way (for example, an event might have type: "hp\_change", target: "team", value: -5 to indicate all team members lose 5 HP).
* store.json and inventory.json: These relate to the optional gold economy. store.json can list items to buy (with prices and effects), and inventory.json would track which items each character has purchased. Since the MVP didn't focus on the item store explicitly, these can be stubbed out or implemented after core features.
* game\_actions.json: serves as a log or history of game events (XP awarded, HP lost, etc.) with timestamps. Each entry might include what happened, to whom, by whom (e.g., teacher awarded 10 XP to Alice at time T). This is useful for review or debugging. It’s appended to, rather than overwritten, to keep a running history (Development Guidelines for JSON-Based -Style Web App.docx).

This directory structure cleanly separates concerns: **backend logic** vs **frontend UI** vs **data storage** vs **tests**. It also aligns with the guidance (backend and frontend separated, data in its own folder) (Development Guidelines for JSON-Based -Style Web App.docx) (Development Guidelines for JSON-Based -Style Web App.docx). Each module/file has a focused responsibility which makes it easier to maintain. For example, if a bug is found in XP calculation, we know it’s likely in xpSystem.js or how it’s invoked, so we can go straight there. Or if data isn’t saving, we inspect the data/ folder or data service functions.

Throughout development, we will enforce this structure. The Git repository will reflect this layout. We’ll also add a .gitignore to exclude any local environment files or perhaps the data/\*.json if we want to avoid committing actual game state (though we might commit a default set of JSON files with sample data for out-of-the-box functionality).

**4. Strategy for Local API Simulation**

To have the frontend communicate with the backend in an offline environment, we will simulate a typical client-server API entirely on the local machine. The two main approaches are: using a local HTTP server (Express) with RESTful endpoints, or using Electron’s IPC mechanism for direct function calls. We plan to use an **Express-based local API** as the primary solution, as it cleanly separates the front-end (running in a browser) from back-end logic and easily supports multiple client devices.

**Express Local Server:** The Express server (server.js) will listen on a local port (e.g., 3000) on the Windows machine. All API routes will be under a path like /api/.... The React frontend, when opened in a browser (likely served from the same server), will use JavaScript fetch() calls to these endpoints to get or modify data. This mimics the way a web app would talk to a cloud server, except here the “server” is just the teacher’s PC. For example, when the app needs to add XP to a character, it calls POST http://localhost:3000/api/characters/:id/xp – Express receives that request, updates the JSON, and responds with the updated character data. Because the endpoints are all local, latency will be very low and no internet is required. The design document confirms that the frontend will use fetch to call local APIs exposed by the Node backend (Development Guidelines for JSON-Based -Style Web App.docx). We will implement all the routes mentioned earlier in Step 4 of the roadmap (XP, HP, power usage, status fetch, team fetch, quest fetch/advance, etc.) (Development Guidelines for JSON-Based -Style Web App.docx) (Development Guidelines for JSON-Based -Style Web App.docx). Each route simply triggers the corresponding game logic and returns a JSON response (usually the updated resource or a success message).

One benefit of using Express is that multiple clients can connect. In a classroom scenario, the teacher runs the server and students connect over the local network via their browsers. The Express approach naturally supports this: students’ devices would make requests to http://<teacher-ip>:3000/api/... for data. We must ensure the server is configured to listen on the correct network interface (e.g., 0.0.0.0 if we want external devices to connect, not just localhost). Security isn’t a major concern on a closed network, but we can implement simple checks, such as requiring a query param or simple token for teacher-only actions, to prevent students from calling teacher APIs (security through obscurity at MVP level, or a basic login session check).

**Electron IPC (Alternative):** In the event we package the app with Electron for single-computer use, we might not need an HTTP layer at all. Instead, the renderer process (which would load the React app) can directly call backend functions via Electron’s inter-process communication. For example, when the user clicks “award XP” in the UI, the front-end could trigger an IPC event like ipcRenderer.send('addXp', { characterId, amount }). The main process (Node) listens for 'addXp' and then calls the xpSystem function, updates JSON, and maybe sends an IPC message back to the renderer with the result. This avoids overhead of HTTP and is purely in-app. However, this approach only works when running inside Electron; it wouldn’t allow separate client devices to connect. Given that our goal includes mobile browser access, the IPC method will be used only in the packaged desktop app scenario, not for general classroom network use. We will design the system so that the core logic functions can be called either via Express routes or directly, to accommodate both modes. In fact, our backend modules can be required by either the Express route handlers or an Electron main process, demonstrating code reuse.

**Simulating API without Express:** Another possible approach (though we likely won’t use since Express is straightforward) is to skip defining real HTTP routes and simply call the backend functions directly from the front-end code when on the same bundle. For instance, one could bundle the backend logic into the front-end (since it’s all JS) and call it, writing to files directly. This would achieve offline operation but tightly couples front and back ends and complicates multi-user usage. The project guidelines mention we could simulate API calls with local function calls if not using Express (Development Guidelines for JSON-Based -Style Web App.docx), but since we aim for a clean separation, we’ll stick with an actual local API server for clarity and extensibility.

**Refreshing Data / Real-time Feedback:** As mentioned, because everything is local, we do not necessarily require real-time web socket connections. We can rely on quick REST polling or user-triggered refreshes. The app can poll key endpoints (like a student’s status or team status) every few seconds to update the UI with any changes (Development Guidelines for JSON-Based -Style Web App.docx). This is a simple strategy to simulate real-time updates without the complexity of setting up WebSocket connections or a pub-sub mechanism. If we later find the need for truly instant updates (perhaps if the teacher triggers a dramatic event and wants all student screens to update immediately), we can implement a WebSocket on the Express server to push notifications. But that is an enhancement (the spec even lists real-time via WebSockets as optional for Phase 2) (Local Gamified Classroom Web App.docx). For MVP, periodic fetch calls are sufficient and easier to implement.

**Error Handling:** The local API should handle errors gracefully. For example, if a power use is requested but the character doesn’t have enough AP, the backend might respond with an error status and message. The front-end should handle that (e.g., show a toast “Not enough AP”). We will design the API responses in a simple JSON format, e.g., { success: false, error: "Insufficient AP" } for an error, and { success: true, data: ... } for success with some data. Being consistent here will make front-end integration easier.

In summary, the strategy is to mimic a standard web app architecture (client-server) but entirely locally. Using Express for a local API provides a familiar development pattern and aligns with the requirement that “All endpoints affect only local JSON” (Development Guidelines for JSON-Based -Style Web App.docx). It will be straightforward to develop and debug (we can use browser dev tools and Postman on localhost), and it keeps the option open to eventually deploy in a networked environment (like the classroom LAN) without any code changes.

**5. Safely Reading/Writing JSON Files**

Using JSON files as a data store introduces considerations for data consistency and safety, especially when multiple actions occur concurrently. Unlike a database which handles concurrent transactions, our file writes could conflict if not managed properly. Below is our strategy to safely read and write JSON files, ensuring data integrity:

**Single Process Access:** Firstly, we will ensure that only our Node server process accesses the JSON files. There will be no direct editing of these files while the app is running, and the Node process itself will be the single writer. This avoids the complexity of inter-process locking (which would be needed if, say, multiple apps or processes tried to edit the files simultaneously). Even within a single Node process, however, we can have multiple asynchronous operations trying to read/write files at the same time (for example, two API calls arriving in quick succession). Node’s fs module does not automatically queue writes – if we call fs.writeFile twice, both operations could overlap. Therefore, we need an in-process coordination mechanism.

**Write Queue / Lock:** We will implement a simple **write queue** system. Essentially, we create a utility that all file write operations go through. If a write is already in progress, subsequent writes will wait (be queued) until the first completes. This can be done by maintaining a Promise chain or using something like a mutex. For example, we could have a global fileWriteLock promise. When a write request comes in, we append a .then() to that promise chain to perform our write after the previous one. This ensures writes happen one at a time in sequence, preventing race conditions. Another approach is using a dedicated npm library for file locks. The spec explicitly mentions using locking or queueing to avoid race conditions (Local Gamified Classroom Web App.docx), so this is a requirement we will fulfil. There are libraries (like proper-lockfile or file-locking) that can lock a file for the duration of a write. However, since our scenario is one Node process, a simple in-memory lock (via a flag or queue) is sufficient and easier to implement.

**Atomic Writes:** We will also ensure that writes to the JSON files are atomic to prevent corruption. One common technique is to write to a temporary file and then rename it to the target filename. This way, if the process crashes mid-write, the original file is untouched. Node’s fs.writeFile is atomic on many systems if the file is small, but to be safe we might do:

1. Write JSON string to filename.tmp
2. fs.fsync (ensure it's flushed to disk), then fs.rename filename.tmp to filename.json. This ensures either the old or new file is present at any time, and no partial data.

Alternatively, since our data volumes are not huge, using fs.writeFileSync may be acceptable for simplicity. Synchronous writes block the event loop, but they guarantee the write is finished when the function returns. Because Node is single-threaded, using synchronous file writes inherently serializes them. For instance, if two nearly simultaneous events trigger writes, the first writeFileSync will finish completely before the second begins (the second can only start after the first function returns). This is a straightforward way to avoid interleaving. The downside is if a write takes a noticeable time, it freezes the server briefly. Given JSON file sizes will likely be small (a few KB or tens of KB for a class worth of data), this might be an acceptable trade-off for MVP simplicity. We can document this decision and later optimize to async writes with a queue if needed for performance.

**Read Strategy:** Reading from JSON files is less problematic but still something to design. We have a choice: read from disk every time we need data, or cache data in memory. For simplicity and consistency, we might read the file at the moment we need it (ensuring we get the latest data including any recent writes). Node’s file system reads are fast for small files, so this shouldn’t be a bottleneck initially. However, repeatedly reading the same data can be redundant. We may implement a basic caching layer where each file’s content is cached in a variable when first read, and we update that cache whenever we write. Then reads can return the cached copy. But we must be careful to update the cache on writes (which we will since we control all writes through our code). This hybrid approach (read cache, write-through on change) can reduce disk I/O and still maintain consistency. Regardless, we’ll hide this behind the data access utility so the rest of the code doesn’t worry about whether it’s from cache or disk.

**Concurrency Example:** To illustrate, consider two students using a power at the same time that affects the same team’s HP. This will result in two nearly simultaneous API calls to deduct HP (via hpSystem). If we handle one request and start writing the updated characters.json while the second is processed, the second might still be reading the old file. Our queue/lock ensures that the second write will wait. But what about the read for the second operation? If the second operation reads characters.json before the first write finishes, it will get stale data. To avoid this, we might lock at a higher level: lock the entire operation (read-modify-write) for a given resource. In practice, such collisions are rare in a classroom scenario, but it’s possible. To handle it robustly, we can use locking not just on file writes but on logical entities. For simplicity, we might accept the risk of minor race conditions in MVP and mitigate by design (e.g., the chance of two students hitting the same second is low). If needed, one approach is to funnel certain simultaneous actions through the teacher (i.e., teacher triggers events, which are sequential). However, an ideal solution is to use a promise-based lock around the game logic function if it involves read-modify-write. This can be custom-coded or using a library. A Stack Overflow discussion affirms that a locking mechanism is needed if multiple processes (or concurrent operations) access a file ([unix - Locking a file for read/write with Node.js - Stack Overflow](https://stackoverflow.com/questions/39988997/locking-a-file-for-read-write-with-node-js" \l ":~:text=1)), so we are taking this advice seriously.

**File Backup:** Since all data is stored locally, we should encourage regular backups. Part of safety is not losing data. We can implement a simple backup system: for instance, each time the app is started, it could copy the current data/ folder to a backup location (timestamped), or the teacher can manually back up by copying files. This isn’t exactly “locking or queuing”, but it is relevant to data safety. We’ll document backup procedures in the deployment guide.

In summary, our data access strategy is:

* Use Node’s fs (no external DB) for offline compatibility ([Reading and writing JSON files in Node.js: A complete tutorial - LogRocket Blog](https://blog.logrocket.com/reading-writing-json-files-node-js-complete-tutorial/#:~:text=Node.js%20provides%20built,specifically%20for%20working%20with%20files)).
* Serialize write operations to prevent concurrent write issues (either via synchronous writes or an explicit queue) (Local Gamified Classroom Web App.docx).
* Consider each read-modify-write as a transaction and guard accordingly.
* Use atomic write practices to avoid partial writes.
* Test concurrent scenarios to ensure no data loss or corruption. By carefully handling file I/O, we maintain the benefits of a lightweight JSON store without falling prey to race conditions or corrupt data.

**6. Testing Approach**

Testing is critical to ensure the app’s game mechanics work correctly and that new changes don’t introduce regressions. We will employ both **unit testing** and **integration testing**, focusing on the core logic (XP/HP/AP rules, powers, quests) and the API endpoints. The testing strategy includes:

**Testing Frameworks:** We plan to use **Jest** as our primary testing framework. Jest is a popular choice for JavaScript projects because it can handle both Node.js tests and React component tests in one integrated tool. It also has built-in mocking and coverage reporting. For testing React components, we will use **@testing-library/react** (React Testing Library) which encourages testing components in a user-centric way (interacting with them as a user would). If we need to simulate browser APIs (like fetch), we can use Jest’s mocking or bring in libraries like msw (Mock Service Worker) to simulate API responses in front-end tests.

**Unit Tests (Game Logic):** We will write unit tests for each backend module:

* *xpSystem tests:* Check that adding XP increases the character’s XP and that if a character’s XP exceeds the threshold for a new level, the level increases and excess XP carries over (if we implement leveling). Verify that removing XP (if implemented for penalties) does not drop below 0 and that logs are written. Also test edge cases like adding 0 XP (no change) or huge XP values.
* *hpSystem tests:* Verify that deducting HP decreases the HP and triggers a "down" state at 0 HP. If a character falls, ensure team consequences are applied (this might require mocking or integrating teamSystem in the test). Also test healing logic if present (HP should not exceed max).
* *apSystem tests:* Ensure that using AP deducts the correct amount and that it refuses if not enough AP. If AP regen function exists, test that it doesn’t exceed the maximum allowed AP.
* *powerSystem tests:* This is more complex: we’ll simulate various scenarios of using powers. For example, test that a Warrior’s “Protect” power actually transfers damage – you might simulate Character A using Protect on Character B, then Character B gets hit, and check that Character A’s HP went down instead by the appropriate fraction. Another test: a Mage tries to use a power without enough AP should result in an error/exception. We might need to stub some data for powers (like a dummy powers.json entry) to test the logic in isolation.
* *teamSystem tests:* If a team consequence function exists, test that when one member falls, others lose the specified HP. Or if a team reward function exists, test distributing XP correctly.
* *questSystem tests:* If we have quest logic (like advancing steps), test that advancing beyond the last step marks quest complete and gives rewards, that you cannot advance if requirements aren’t met, etc.

These unit tests focus on the pure functions without needing the server. We can mock file reads/writes by injecting test data. For example, pass a character object to xpSystem.addXp rather than having it read from the file. This may involve slight refactoring to allow dependency injection (or simpler, we call the function then manually load the file to see if it wrote correctly – but writing to actual file in unit tests is not ideal). A better approach is to abstract the data layer so we can substitute an in-memory data store during tests.

**Integration Tests (API):** Using **Supertest** (a Node library for testing Express apps) or even Jest’s fetch with a running server, we will test the API endpoints end-to-end. These tests will start the Express server (or better, use an instance of the Express app) and perform HTTP calls:

* Test that POST /api/characters/:id/xp actually increases the XP in the data and returns 200 OK with updated data. Possibly verify the JSON file or the response content to ensure it matches expectation.
* Test that POST /api/characters/:id/power fails with a 400 error if AP is insufficient.
* Test that a student role cannot call a teacher-only endpoint (if we implemented role checks, e.g., perhaps POST /api/characters/:id/xp might require teacher authentication — for testing, we might simulate by providing a session token or by calling a lower-level function if not implemented).
* Test quest progression: create a test quest in quests.json, then call advance and see that the character’s quest status updated.
* If the app has login, test that login with a student vs teacher yields proper responses or sets session (though likely our MVP login is simple without session handling).

Integration tests ensure that our server, logic, and file IO all work together correctly. These tests will likely manipulate the actual JSON files in the data directory. To avoid messing up real data, we will use a separate set of test data files (maybe copy the data folder to data\_test before tests run, and point our app to use that). Alternatively, before each test, we can programmatically write fresh JSON content (known values) to the files, run the API calls, and then verify the outcome.

**Front-End Tests:** We will write a few key tests for the React front end. For instance:

* Render the DashboardView with a sample character object and ensure that the XP, HP, AP values are displayed correctly (this can be done by providing a test context or prop data).
* Simulate a user clicking a "Use Power" button and verify that the appropriate function (mocked fetch) is called. We can mock the global fetch in tests to immediately return a success response, then ensure the component updates state (perhaps by checking that the AP value decreased after the click).
* Test that the TeacherView lists all students given some sample data, and that clicking an "award XP" button calls the API function.

These tests help catch UI bugs (like a button not wired to any action, or a piece of state not updating). They complement the backend tests by ensuring the front-end is correctly integrated.

**Test Coverage Focus:** We will prioritize coverage on **game logic and rules**, as well as **critical API flows**. The reason is that the educational value and fairness of the game rely on these rules working properly (we don’t want a bug that gives someone infinite XP or fails to penalize HP when it should). The MVP features list (character stats, powers, quests, team effects) guides our testing focus (Minimum Viable Product (MVP) Definition.docx) (Minimum Viable Product (MVP) Definition.docx). For instance, the team shared HP loss is a key mechanic to test, as is the proper unlocking and AP costing of powers. We also want to test that all data modifications persist (i.e., if we award XP and then restart the server, the XP gain was indeed saved to the JSON file). That can be tested by checking the file content after an operation.

**Testing Tools & Automation:** Besides Jest, we will integrate **ESLint** for linting the code. Linting helps catch bugs like undefined variables or inconsistent usage before they even run. It also enforces code style, making contributions more uniform. We can run lint checks as part of the test suite (for example, have an npm script that runs eslint . and then jest).

We will set up the test scripts in package.json such that running npm test will run the full test suite. We aim for a good percentage of coverage (perhaps 80%+ of critical files). We will also consider edge cases and failure cases in tests, not just the happy paths.

**Continuous Testing:** In our GitHub CI (next section), we will configure it to run tests on each push. This way, automated tests act as a safety net for ongoing development. As the Dev.to article says, having a workflow run unit tests and lint on each push acts as a gatekeeper to catch issues early ([How I Set Up a GitHub Workflow to Automatically Lint and Test My Vue Project on Push - DEV Community](https://dev.to/juniordevforlife/how-i-automatically-lint-and-test-my-vue-project-on-push-in-github-4nnh#:~:text=can%20see%20in%20the%20repo%2C,share%20this%20workflow%20with%20you)). We will use that philosophy to maintain code quality.

**Manual and User Testing:** In addition to automated tests, we acknowledge the importance of manual testing for user experience. We will do test runs of the app in a scenario close to real use: one instance as teacher, multiple browser windows as different students (perhaps using different browsers or private mode to simulate different devices). We’ll walk through a class session scenario: teacher gives points, triggers an event, students use powers, complete a quest, etc., to see that everything syncs up and behaves. This can reveal integration issues that pure automated tests might not (e.g., if the UI isn’t updating fast enough, or a certain sequence of actions leads to an unexpected state).

By combining these testing strategies, we aim to cover both the correctness of the game logic and the reliability of the overall application. Automated tests will give us confidence for each code change, and manual tests will ensure the app feels solid in practice.

**7. Deployment Guide (Local Windows Deployment and Optional Electron)**

Deploying the application locally on a Windows machine will involve setting up the Node server and ensuring the front-end is accessible to users (students) in the classroom. We also outline how to bundle the app with Electron for an even simpler desktop experience, though that is optional.

**Prerequisites:** On the target Windows PC (likely the teacher’s laptop or a classroom computer), install the latest LTS version of Node.js (which includes npm). Also, if not using Electron, ensure a modern web browser is available (Chrome, Firefox, or Edge) for the teacher to open the app, and that student devices have browsers and can connect via the local network (if applicable).

**Deployment Steps (Without Electron):**

1. **Obtain the Code:** Clone the GitHub repository onto the Windows machine. This could be done by installing Git and running git clone <repo-url>, or by downloading a ZIP of the project and extracting it.
2. **Install Dependencies:** Open a PowerShell or Command Prompt in the project directory and run npm install. This will download all required Node packages for both backend and frontend. (If the frontend is a separate project, you might need to cd frontend and run npm install there as well, but we can also configure a root-level install that covers both).
3. **Build Frontend:** Run the front-end build script to create an optimised production bundle. For example, npm run build (common for Create React App) which outputs static files (HTML, JS, CSS) into a frontend/build or dist folder. Ensure this build completes successfully.
4. **Configure (if needed):** By default, we may have the app using a certain port (say 3000). If that port is in use or needs to change, adjust the configuration (maybe in an .env file or in server.js). Also, ensure that the data/ folder contains the necessary JSON files. For first run, it might have sample data or mostly empty arrays. The teacher can pre-populate some info (like creating user accounts for students) either by editing JSON or via the app’s interface once running.
5. **Start the Server:** Run npm start or specifically node server.js. The Express server will start. By default, it will likely listen on port 3000 (so URL is [http://localhost:3000](http://localhost:3000/)). The server should also serve the built front-end, so navigating to the root URL in a browser will load the React app.
6. **Access the App (Teacher):** On the same machine, open a browser and go to http://localhost:3000. The login screen should appear. The teacher can log in as teacher (the exact mechanism depends on our implementation; it might be selecting a teacher user from a list). Now the teacher can use the interface.
7. **Access the App (Students):** For students to connect using their own devices, the teacher’s machine and students’ devices must be on the same local network (e.g., same Wi-Fi or LAN). The teacher’s machine might have an IP like 192.168.1.100. Students can then open their browser and navigate to http://192.168.1.100:3000 (assuming default port 3000) to load the app. Since the app is not internet-hosted, the students need that local network connection. It’s offline in the sense of no external internet, but a local network is used. If the class lacks networking, an alternative is to have students use the teacher’s machine in turns, or set up a local hotspot. In most cases, a simple Wi-Fi router or the teacher’s laptop acting as a hotspot can allow the connections.
8. **Firewall Considerations:** Windows may pop up a firewall alert when the Node server starts listening. The teacher should allow access on private networks so that student devices can connect. This is important; otherwise, the connections from student browsers might be blocked.
9. **Using the App:** Once connected, students can log in (we will have provided some way, perhaps each student selects their name from a list which corresponds to a user in users.json). The teacher and students can now interact through the app in real time. All data changes (XP, HP adjustments, etc.) are being saved to JSON on the teacher’s machine as they happen.
10. **Shutting Down:** To stop the app, the teacher can close the Node server (Ctrl+C in the terminal). All data is preserved in the JSON files. On next start, the state continues from last saved values.

By following the above, deployment is essentially **running a local Node server**. The spec describes this scenario: running on a local server (teacher’s laptop) with browser-based frontends for students on the same network, and data remaining local/offline (Local Gamified Classroom Web App.docx). We have achieved exactly that.

**Electron Deployment (Optional):** If we package with Electron, the deployment becomes a bit different:

* We would create an Electron main process script that launches a browser window and either runs an internal express server or directly serves the React app from memory.
* Using a tool like Electron Builder, we can create an installer or standalone .exe for Windows. The teacher would install this like a normal application.
* When launched, the Electron app could show the teacher interface directly. We would still need students to connect via a browser if we want multi-user play. This can be done by having the Electron app also start an Express server in the background listening on a port. Electron’s main process can incorporate our existing server.js. Essentially, the Electron app is a wrapper that does two things: opens a window (pointing to localhost:3000 or an internal file URL) and starts the server.
* This way, the teacher just double-clicks the app, the window opens with the UI, and students can connect as before via the teacher’s IP. Electron handles packaging Node and Chromium together, so it might be a large binary but user-friendly.
* Alternatively, for a purely single-computer usage (no student devices connecting, like one shared screen), Electron would not require any network at all – the game would be played on one machine.
* We will document how to build the Electron package if we provide it: likely npm run electron-build after installing needed devDependencies.

**Maintenance and Running:** For everyday use, the teacher should periodically update the app if there are new versions (pull from GitHub or reinstall the Electron app). If the teacher is not code-savvy, the Electron route might be easier (just use the app UI). If using the Node approach, we could provide a simple double-click script to start the server (like a start.bat that runs node server.js to save them from the command line).

**Troubleshooting:** We will include common troubleshooting tips:

* If students cannot connect, check firewall or that they are on the same network.
* If the app is not loading, ensure the server is running and the browser points to the correct IP/port.
* If data seems wrong, one can inspect the JSON files in the data folder to verify what’s stored.
* Backup the data folder regularly (just copy it somewhere safe) especially before upgrading to a new version of the app.

By following this guide, a teacher should be able to get the app running locally without internet. The key is that it’s relatively straightforward: install Node, run a couple commands, and use the browser. For a more polished distribution, the Electron packaging can simplify usage at the cost of an initial packaging effort. In either case, the deployment does not rely on any external services, fulfilling the offline requirement.

**8. Continuous Integration (CI) Recommendations for GitHub**

Using GitHub for version control enables us to set up Continuous Integration workflows that run automatically on each commit or pull request. This helps maintain code quality and catch issues early. Here are the CI practices we will implement or recommend:

**GitHub Actions Workflow:** We will create a CI workflow (YAML file in .github/workflows/ci.yml) that triggers on pushes and pull requests to the main branch. The workflow will run on a Windows or Linux runner (Node runs on both; Linux is faster on GitHub Actions typically, but we might include one Windows run to catch OS-specific issues). We will specify Node.js latest LTS in the setup. The steps of the workflow will include:

* **Checkout code:** Use actions/checkout@v3 to pull the repo code.
* **Set up Node:** Use actions/setup-node@v3 to install Node (e.g., with node-version: 18).
* **Install dependencies:** Run npm ci to install packages (ci is preferred for CI for a clean install).
* **Run lint:** Run npm run lint (assuming we have an npm script for ESLint). This will check for any code style or potential error issues. If lint finds issues, the job will fail. This ensures all code merged follows our lint rules.
* **Run tests:** Run npm test which will execute our Jest test suite. This covers unit and integration tests. If any test fails, the CI fails.
* **Build (optional):** We might also run npm run build to ensure the front-end builds successfully (and maybe even run a minimal server start to catch any runtime issues). This step ensures that our code changes didn’t break the build process.
* **Artifacts/Coverage (optional):** We can configure Jest to output a coverage report and have the CI upload it as an artifact, or use a service like Coveralls. This helps track test coverage over time. Not mandatory, but a nice addition.

This automated workflow essentially replicates the steps a developer would do locally (install, lint, test, build). It's similar to the example of a GitHub workflow that runs tests and lint on each push ([How I Set Up a GitHub Workflow to Automatically Lint and Test My Vue Project on Push - DEV Community](https://dev.to/juniordevforlife/how-i-automatically-lint-and-test-my-vue-project-on-push-in-github-4nnh#:~:text=can%20see%20in%20the%20repo%2C,share%20this%20workflow%20with%20you)) – acting as a gatekeeper to ensure code quality. If the CI fails, we will not merge the changes until fixed.

**Branch Protection:** We will enable branch protection on the main (or master) branch in GitHub, requiring that the CI workflow passes before any pull request can be merged. This means every code change must go through the CI and meet our standards. This protects the main branch from broken code.

**Code Review & PRs:** Even with CI, we will use GitHub’s Pull Request system for code review. All substantial changes should be done in feature branches and submitted as PRs. Team members (or even the teacher if they are involved in development) can review the code, discuss, and approve. CI results will be visible on the PR, giving confidence that the change doesn’t break anything.

**Linting and Formatting:** In addition to running ESLint in CI, we recommend using a pre-commit hook (with a tool like Husky) during development to lint or prettify code before committing. This reduces the chances of style issues making it to the remote repo. The CI lint step is a backstop to catch anything missed.

**Dependency Management:** We might use a tool like Dependabot (which GitHub provides) to automatically open PRs for dependency updates. The CI will test those updates to ensure they don’t break the app. This keeps the project secure and up-to-date.

**CI for Multiple Environments:** We could configure the workflow matrix to test on multiple Node versions (e.g., Node 16 and 18) to ensure compatibility, but since this is a closed environment app, sticking to one known Node version is fine. Testing on Windows runner could be useful given the deployment target is Windows, to catch any filesystem path issues or line-ending issues. For example, ensure that our file paths or any OS-specific code works on Windows (like file locking behavior).

**Continuous Delivery (CD):** While not explicitly required, if we get to packaging releases, we could use GitHub Actions to build the Electron app and perhaps attach binaries to GitHub Releases. This would automate the creation of new version installers whenever we tag a release. For now, this is optional and can be done later if needed.

**Monitoring CI:** We’ll add a status badge in the README showing the CI status (passing/failing) for transparency.

By maintaining this CI setup, every push is verified. This reduces the chance of deploying something broken to the teacher’s machine. It also makes onboarding new contributors easier, as they get immediate feedback if something is amiss in their code. As GitHub’s own guide notes, a CI workflow to build and test Node.js code ensures that if tests pass, one can confidently deploy or publish the code ([Building and testing Node.js - GitHub Docs](https://docs.github.com/en/actions/use-cases-and-examples/building-and-testing/building-and-testing-nodejs#:~:text=This%20guide%20shows%20you%20how,code%20or%20publish%20a%20package)). In our case, deployment is manual to the classroom machine, but knowing the commit passed CI gives confidence in its stability.

In conclusion, this implementation and development plan leverages a modern tech stack suited for offline use, outlines a clear development progression from MVP to full product, and addresses practical concerns like data safety, testing, and deployment. By following this plan, we can develop a -inspired classroom RPG app that meets the requirements: fully offline, JSON-backed, Windows-deployable, mobile-friendly, and maintainable via GitHub and CI. Each aspect of the plan ties back to the project’s goals and constraints, ensuring we stay within scope (no external databases or internet needed (Development Guidelines for JSON-Based -Style Web App.docx)) while delivering a functional and engaging gamified classroom experience.