ChatGPT

Quickstart Summary – *Blueprint for Building an RPG Virtual Tabletop (VTT)*

- **Project Setup:** Use **React 18 + TypeScript** with **Vite** for fast bundling. Scaffold a responsive app with a **left-side vertical tab bar** (for Map, Character, Spells, Skills, Fighter). Integrate **React Router** for nested routes (e.g. `/game/:roomId/map`) and set up a global **state management** solution (e.g. Zustand for simple global state plus XState for complex flows [1] [2]). Configure **Tailwind CSS** (utility classes) and adopt a component library (e.g. Shadcn UI built on Radix) for accessible, styled UI elements [3].

- **Design Foundation:** Establish a **design system** with dark-theme-friendly colors, readable typography, and consistent spacing. Build reusable components (buttons, modals, tooltips, form inputs, icons) and ensure high contrast and keyboard navigability (WCAG 2.1 AA compliance). Implement the left tab navigation (vertical icon bar) with focus indicators and keyboard shortcuts (e.g. <kbd>Alt+1..5</kbd> to switch tabs). Test the layout on various screen sizes (desktop and tablet) – it should collapse or scroll as needed for smaller screens.

- **Backend & Data:** Set up a simple **Node.js backend** (Express + tRPC or REST) with **WebSocket (Socket.io)** for real-time events and **Prisma + PostgreSQL** for persistence. Design relational **data models** for all game entities: User accounts, Game Room sessions, Characters (with Stats, Inventory, Skill choices), Items, Spells, SkillTree nodes/edges, Map POIs, Tokens (map units), Encounters, Turn order, Actions, Conditions, and AuditLogs. Use Prisma schema to define tables and relations (e.g. Character has many Items, Room has many Users with a role). Start with a **mock server** or stubbed data for frontend development, then implement real DB operations and secure APIs. Ensure **auth** via session cookies or JWT, with role-based checks (GM vs Player) on server for sensitive actions.

- **Map Canvas Implementation:** For the **Map tab**, integrate a high-performance 2D rendering library (**PixiJS via React-Pixi** or **Konva**). Display a large world map (as an image or tile layer) with smooth **panning and zooming** (mouse drag, scroll/pinch to zoom). Maintain a camera transform (world coordinates vs screen) to handle zoom levels. Plot interactive **markers/POIs** for towns, dungeons, etc., and implement **clustering or level-of-detail**: e.g. at far zoom, combine nearby markers or hide labels to reduce clutter. Provide optional overlays: a toggleable **grid** (draw grid lines or hexes over the map), **fog of war** (mask unrevealed areas, updated by player token vision), and a **measuring tool** (drag to measure distance on the map). Use layering (e.g. separate canvas layers or Pixi containers) for terrain, tokens, and annotations so they can be toggled or z-indexed easily. Optimize rendering by culling off-screen elements and using spatial indexing for hit-testing clicks on markers. Target a steady **60 FPS** for common interactions by leveraging GPU acceleration (e.g. Pixi/WebGL) [4].

- **Character Sheet Tab:** Build a **Character Sheet** UI reminiscent of Shakes & Fidget's style (character portrait, stats, gear slots, backpack) [5] [6]. Display **base attributes** (e.g. Strength, Agility, Intelligence, etc.) and **derived stats** (HP, defense, damage, etc.), updating derived values automatically when bases or equipment change. Implement a **stat allocation** interface for level-up: allow players to spend available stat points on attributes with "+" buttons (disable or hide when no points). Include **validation** (e.g. cannot exceed available points, confirm allocation). Below stats, render **equipment slots** (head, chest, weapon, etc.): each slot shows the item icon or is empty. On the side, show the **Inventory** grid (e.g. 20 slots) with item icons. Enable **drag-and-drop** or contextual actions to equip items from inventory into slots and to unequip back to

inventory. Enforce item requirements (level, class, etc.) – if not met, show an error or prevent the drop. When equipped, update the character's stats (e.g. add item bonuses) and remove the item from backpack. Show **item tooltips** on hover: name, stats, rarity (color-coded), and any effects. Manage inventory sorting or filtering (optional: allow sorting by item type or rarity). Ensure all these interactions also trigger state updates and backend events (so other players or the GM can see changes, if applicable). The character tab should be optimized to avoid re-rendering the whole page when one stat changes (use memoization or granular state slices).

- **Spells & Abilities Tab:** Implement a **Spells Library** interface for the character's known spells. Provide filtering and sorting controls (e.g. filter by school, type of spell, mana cost, range, target type) to quickly find spells. List spells with concise details (name, level or cost, brief effect) and allow the player to select spells into an active **loadout** (for instance, "prepared spells" or quick-access slots). If there's a limit to prepared spells, enforce it and visually indicate slots filled vs available. When a spell is selected, show a detailed panel with full description: effects, damage or healing formula, range/area, resource cost, cooldown, and any special conditions (e.g. "requires line of sight" or "stuns target"). If applicable, show whether the spell is currently **on cooldown** or available – e.g. a cooldown timer or dimmed icon. Ensure the UI for selecting/unselecting spells updates the state and is sent to server (with optimistic feedback). In preparation for combat integration, structure the data so that casting a spell can be invoked with an event (target selection happens in the Fighter tab, but the availability and definitions come from this data). The spells tab should handle a large number of spells gracefully (use virtualization if the list is long, or pagination by spell level).

- **Skill Tree Tab:** Create a **skill tree** (talent web) interface inspired by Path of Exile's passive tree (but much smaller). Represent the **graph** of nodes and edges: each **SkillNode** has properties like an ID, name, description, and possibly an icon/effect (e.g. +5% crit chance), plus references to prerequisite nodes. **SkillEdge** records connections between nodes (or infer edges by a parent/child mapping). Use a library or custom SVG/canvas to render this graph: **React Flow** is a strong option, providing pan/zoom and interactive nodes out-of-the-box [7]. Lay out the nodes in a clear manner (e.g. a radial or grid layout) such that prerequisite links are visible lines. Implement **zooming and panning** so users can explore the tree if it doesn't fit on one screen. Each node should visually indicate its status: unlocked (invested), unlockable (prerequisites met and player has a point), or locked. Show tooltip or detail on click (what the node gives). Allow the player to **spend skill points** (earned via level-ups or quests) to unlock a node: on clicking an unlockable node, if the player has points available, mark it as unlocked (and send an update to backend). Deduct the point and **unlock any newly reachable nodes** (e.g. highlight adjacent nodes now available). Implement **respec (refund)** rules: e.g. if the game allows respec, provide a way to unlearn nodes (perhaps only the most recent ones or for a cost). Ensure that a node cannot be removed if it's a prerequisite for another node still taken (enforce dependency ordering on refund). Possibly include a "Reset All" button for GM or players (with permission) to reset the tree, if that's allowed. Keep the graph rendering performance high by only re-rendering the changed nodes (libraries like React Flow handle this efficiently). Make sure the design is color-blind friendly (different shapes or labels for node states, not just colors).

- **Fighter (Combat) Tab:** Develop the **tactical combat interface** for turn-based encounters. This will reuse the **map canvas** but focused on a specific battle map (could be the same world map zoomed into an area, or a separate grid map for the encounter). When a GM **starts an encounter**, initialize an **Encounter** data object containing participating tokens (PCs and NPCs), their initiative order, and status. Display an **initiative tracker** UI (often a sidebar or top bar): a sorted list of combatants (by initiative roll) with highlights on the current turn. Allow the GM to manually set or adjust initiative values (in case of custom ordering or delays). On **encounter**

start, each client in the room gets a "combat start" event with the turn order. Now, enable **turn-based controls**: highlight the active character's token and perhaps center the map on them. The active player (or GM if controlling NPC) can perform actions on their turn. Key actions to implement: **Movement** – clicking or dragging their token on the grid. Show a **grid highlight or path** as they drag, and enforce movement limits (e.g. 30ft speed = 6 tiles max). If needed, highlight reachable tiles in a different color. Snap token positions to grid coordinates. Include logic for **difficult terrain**: some tiles may count as 2 steps – track movement cost, so if a token enters a difficult tile, decrement remaining movement by 2. Prevent moves that exceed the allowed distance (e.g. stop the drag or indicate not allowed). **Targeting** – allow a player to target another token (e.g. click an enemy to target for an attack or spell). You might show an icon or outline on targeted tokens. For **attacks and spells**, integrate with the data from Character (weapons) or Spells tab: e.g. the player chooses an action ("Attack with sword" or "Cast Fireball") then selects target(s). Implement **AoE template** tools: e.g. a player can choose a "fireball" and then place a circular template on the map to see which tokens would be affected; when confirmed, apply effects to those tokens. The system should handle **applying damage and conditions**: after an attack or spell, update the target's HP (and if HP ≤ 0, mark the token as defeated/dead), and add any **conditions** (status effects) such as stunned, poisoned, etc., with a duration (e.g. expires in N rounds). Visually represent conditions on tokens (small condition icons or color tints, with tooltip for details). Provide a way to **end the turn** (a button for the active user), which then advances the initiative to the next token. Automatically increment the **round** counter when the order cycles back to the top, and handle condition expiration (e.g. decrement duration, remove if 0). Throughout, ensure all players' UIs stay in sync via real-time events (everyone sees the token movements, HP changes, etc.). Include **GM overrides**: the Game Master can move any token (to reposition monsters or even player tokens if needed), adjust HP manually (e.g. apply healing or correct an error), and apply or remove conditions at will. The GM should also have a **"pause combat"** toggle – when active, players' ability to move or act is temporarily disabled (useful if the GM needs to clarify something or prevent further actions). When the encounter is over, a **combat end** action will reset turn order and possibly remove any temporary turn UI. Aim to keep the **map interaction at 30-60 FPS** even with multiple tokens; use optimizations like only re-rendering moving tokens and minimal redraw of static map. Test latency: even with modest network latency, the optimistic updates (moving a token locally immediately) combined with server reconciliation ensure a smooth experience.

- **Real-Time Sync:** Use **Socket.io** (or similar WS library) on the backend to manage real-time communications. Design the server to create a **room** for each game session (`roomId`), and have clients **join** their room channel upon loading a game. Define a clear **event model** for all real-time interactions: e.g. `"playerJoined"`, `"playerLeft"`, `"map.pan"` (if broadcasting a GM map ping), `"token.move"`, `"token.update"` (for any positional or stat changes), `"encounter.start"`, `"turn.next"`, `"action.perform"` (for attacks/spells), `"condition.applied"`, etc. Each event will have a payload with necessary data (e.g. `token.move` carries token ID and new coordinates). Implement **presence** notifications: broadcast when a user connects or disconnects from the room so the UI can show who's online. Use **optimistic UI** for snappy feel – e.g. when a player moves their token, update it on their screen immediately [8], but also emit a `token.move` event; on the server, validate the move (distance, permissions) and broadcast to others. If the server finds an illegal move (e.g. too far or player not allowed), it can either correct the position with another event or refuse and instruct that client to revert (though with good client checks, this should be rare). Handle **simultaneous actions** by queuing or sequencing events: e.g. if two players somehow act at once, the server may enforce an order (especially in turn-based combat, ignore out-of-turn actions or put them in a log). Use unique identifiers or timestamps on actions to implement **last-writer-wins** for conflicting state updates, except that GM commands always override (e.g. if GM moves a token,

that position is authoritative). Implement **acknowledgements** for critical events if needed (Socket.io by default is reliable, but for idempotency we might tag events with an ID and have clients ignore duplicates). Plan for **reconnection**: if a player refreshes or drops, on rejoin the client should fetch the latest game state (e.g. via an API call like `GET /game/:roomId/state` or re-sync events) to catch up on any missed updates while offline. Use heartbeat or the built-in Socket.io pings to detect disconnects promptly so the server can mark users offline and possibly transfer control if a GM leaves. Overall, keep the server as the **source of truth** for game state – clients can simulate for responsiveness but the final say comes from the server to resolve any discrepancies.

- **Security & Roles:** Implement an **authentication** system for user login (even if just a simple email/password or username for now). Use HTTPS for all communications. Use server-side session cookies or JWTs to secure API calls and socket connections (e.g. Socket.io can validate a token on connection). Distinguish **roles**: a **GM (Game Master)** vs **Player** – store role info in the Room or a membership table. Throughout the code, **authorize actions**: e.g. only GM can spawn or delete tokens, only GM can initiate combat or reveal map areas, players can only move tokens they control (their character) and not others, etc. Validate this on the server for every relevant event or API call – do not trust client-side checks alone. Implement basic **rate limiting** on actions that could be spammed – e.g. movement events or spell casts – to prevent flooding (this could be as simple as limiting position updates to e.g. 10 per second per client, and ignoring excess). All user inputs should be **validated/sanitized** on server: e.g. item names or chat messages (if any) should be escaped to prevent XSS, numeric inputs clamped to expected ranges, etc. Maintain an **audit log** of critical actions (in an AuditLog table): e.g. when a player spends stat points, casts a spell, or when the GM uses a special override – log who, what, and when. This not only helps with debugging but also moderation if needed. Plan a basic **abuse prevention**: e.g. invite links to a room should be protected (maybe with an invite code or link that expires), and have the GM able to kick a player if necessary. On the client, hide or disable any GM-only UI for players (based on role), but still **never rely solely on hiding** – the server must check permissions on each action. By covering these security bases, the game will be resilient against casual cheating or mistakes.

- **Performance & Optimization:** Establish performance targets early: aim for **60 FPS** on animation-heavy screens (map and combat) on a typical machine, and **Time to Interactive** within a few seconds on initial load. Use **code-splitting** by route – e.g. don't load the heavy canvas or combat logic until the user enters a game session – to keep the initial bundle small. Continuously profile the app: use React DevTools Profiler to ensure components aren't re-rendering unnecessarily (apply `React.memo`, `useMemo`, etc., especially for lists of items, spells, or map markers). Implement **list virtualization** for long lists (inventory, spell list) so only visible items render. For the canvas, prefer batch-drawing techniques (e.g. Pixi will batch sprites automatically; if using Konva, leverage Layer caching for static elements). If using WebGL (Pixi), manage texture memory: unload or compress assets (images) that aren't in use (for example, if you have multiple map images). Set reasonable limits on map size or number of tokens; if the map is huge, consider tiling it so the client only loads what's visible. Use **throttle/debounce** on expensive operations like recalculating fog-of-war or on resize events. Monitor memory usage for leaks – ensure to cleanup event listeners on unmount, and destroy any Pixi/Canvas objects properly when leaving a page or ending an encounter. Implement **observability** hooks: integrate an error tracking service (like Sentry) to catch JS exceptions and backend errors, and add custom logging for performance issues (e.g. if a frame takes > 50ms to render, log it). Define some **analytics events** to track usage (without PII): e.g. log how long combats typically last, or which tabs are used the most, to identify hotspots or unused features. All these measures will ensure the app runs smoothly for users and that you have insight into its runtime behavior.

- **Testing Strategy:** Set up a robust testing approach for quality. Write **unit tests** for all pure logic modules: e.g. functions calculating derived stats, damage formulas, movement range calculations, cooldown reducers, etc. Use these to verify rules (e.g. difficult terrain cost doubling movement [9], advantage dice logic) are correct. For React components, write **component tests** (with Jest/React Testing Library) for crucial UI logic – e.g. the stat allocation component (simulate adding points, expect the available points to decrement and stat to increment), or the inventory drag-drop (maybe simulate an "equip" action via function call and ensure state updates). Use **integration tests** to cover end-to-end flows: you can use a headless browser via Playwright or Cypress to simulate a user journey – for example, login -> join game -> open character tab -> allocate stats -> equip item -> see stat change -> enter combat -> move token -> end turn, etc. This ensures the pieces work together. Specifically test real-time behavior in a controlled environment: you can instantiate two WebSocket clients in a test to simulate two players and ensure that when one does an action, the other receives the update (for example, moving a token or casting a spell yields the expected state on both sides). Include tests for race conditions or conflict resolution: e.g. simulate two simultaneous stat increases and ensure the server only allows one or handles them gracefully. Use **deterministic seeds** or mock RNG for combat simulations so that tests for an attack always produce the same outcome (or test both branches of hit/miss consistently). Prepare **fixtures** for data – e.g. a sample character JSON, a sample list of spells – to use in tests without hitting the real DB. For the backend, write API tests (with supertest or similar) to verify authorization is enforced (e.g. a player role cannot call a GM-only endpoint). Automate these tests in a **CI pipeline** (e.g. GitHub Actions) so that on every push, the unit/integration tests run. Aim for good coverage on critical logic (rules engine, reducers, etc.), and use **manual testing** for the experiential aspects (like dragging on canvas) combined with user feedback to fine-tune.

- **Deployment & Ops:** Plan for deploying the app in stages. Use separate **environments**: development (local, with perhaps a local Postgres and in-memory or local socket server), a **staging** environment (hosted test server and database, where you can do final testing with a small group), and **production**. For the frontend, consider deploying as a **static bundle** to a CDN (since it's a SPA) – e.g. use Vite to build and host the files on Netlify or Vercel, or serve via Express static if using the same server. The backend (API + WebSocket server) can be a Node process hosted on a service (AWS, Heroku, Fly.io, etc.) – ensure it's configured for SSL (so wss and https). Use a **CDN** for static assets and possibly for heavy media: e.g. if you have large map images or token images, store them in cloud storage and serve via CDN so they load quickly for clients. Set up **automated database migrations** (Prisma Migrate) for schema changes and have a backup strategy: enable daily backups on the Postgres or use a backup script (important to prevent data loss of game progress). Monitoring in production: keep an eye on server CPU/memory (especially if many concurrent WebSocket connections – Node can typically handle many, but track it). Use logging and maybe an APM tool to catch performance issues in prod. Have a **rollback plan** for deployments: since it's real-time, ideally maintain backward compatibility between client and server during deploys (or do coordinated deploys). If a new deploy fails, be ready to redeploy the previous stable version quickly. It's wise to version your API or events if making breaking changes, to avoid crashes if some users haven't refreshed the client. Lastly, set up basic **operational alerts**: e.g. if the server goes down or error rate spikes, you get notified and can respond.

- **Project Timeline & Milestones:** Divide the work into sensible **phases** with milestones. *Week 1:* Foundation – initialize the project (React + TS + Vite), set up Tailwind and verify you can build a simple page. Implement the routing structure with placeholder pages for each tab and the tab bar navigation. *Week 2:* Design system – choose color scheme, build base components (buttons, inputs, modals), and ensure global styles (Tailwind config) are in place. Implement the left tab

bar UI fully. *Week 3:* Map Tab MVP – integrate a map component (start with a simple `<img>` or a small canvas prototype). Implement basic pan/zoom on the map, and plot a few dummy markers. No clustering yet; focus on smooth interaction and event handling. *Week 4:* Character Tab MVP – create a Character context/state. Render character info, a few stats, and inventory with some sample items. Implement stat increase buttons (front-end only) and inventory drag-drop (using a dummy state update). *Week 5:* Backend basic setup – set up Express or tRPC server, connect to a dev Postgres. Define schema for User, Character, Item, etc., and implement minimal API endpoints or procedures to load a character and update stats. Integrate the front-end with these APIs (e.g. loading real character data on login). *Week 6:* Inventory & Equipment – implement backend support for equipping items (e.g. an API call or socket event). Complete the character tab interactions: actually remove item from inventory and assign to slot, with server persistence. Show derived stat recalculation when gear changes. *Week 7:* Spells Tab – create a spells database (or JSON) and render the list in the UI. Implement filtering UI and the ability to mark prepared spells. Hook up an update call so it saves which spells are prepared for the character. *Week 8:* Skill Tree – decide on a rendering approach (e.g. integrate React Flow). Build a small sample skill graph (5–10 nodes) and get pan/zoom working in the UI. Implement unlocking logic on the front-end with validation of prerequisites. *Week 9:* Skill Tree backend – expand the skill graph to the full design (if it's more nodes) loaded from data. Allow skill point assignments to persist (API call). Implement respec on front-end (with simple rules) and persist (this might involve deleting some data in DB or a special endpoint). *Week 10:* Fighter Tab basics – set up the combat UI framework. Reuse the Map component but prepare a mode where tokens are interactive. Allow the GM to initiate an encounter: perhaps a temporary hardcoded list of participant tokens for now. Display turn order UI (static list). *Week 11:* Combat mechanics – implement turn cycling logic (front-end state for current turn). Allow active player to move their token (with range checking). Implement a simple attack action: a button to "Attack" that automatically hits or misses (for prototype, maybe no targeting yet or assume one enemy). Update HP on target and check for defeat. *Week 12:* Combat expansion – implement targeting properly (click target to select). Add support for spell actions in combat: if a spell is prepared, allow using it (reduce a resource or mark on cooldown). Implement condition effects: create a way to apply a "stunned" status on an enemy for X turns and skip their turn in order. *Week 13:* GM Tools – add UI for GM to add/remove tokens in combat (e.g. a plus button to create a monster token at a clicked location on map). Enable GM to override initiative or HP (maybe by clicking an entry in turn tracker to edit value). Implement pause/resume toggle that greys out player controls. *Week 14:* Real-time integration – integrate Socket.io. Test basic messaging between two clients: join room, broadcast a chat or simple "ping". Then wire it into subsystems: e.g. when a token moves on one client, emit event and update other clients' state. One by one, make character updates, spell prep, and combat actions go through the server. This week will involve a lot of testing with multiple clients running. *Week 15:* Polish & Optional Features – implement map marker clustering properly (if many POIs, use a clustering algorithm or simple threshold logic). Add the fog-of-war mask and vision (this can be complex, so start simple: e.g. reveal a radius around player token). Add the measuring tool on map. Refine UI styling (make sure the theme is consistent, add animations or highlighting for active turn, etc.). Ensure accessibility attributes (aria labels on icon buttons, etc.) are all in place. *Week 16:* Testing & Performance – write/finish automated tests for all new logic. Conduct load tests: e.g. simulate 4-5 players in an encounter to see if the UI or server slows down. Use profiling tools to find slow spots (maybe large state updates) and optimize them (e.g. prune unnecessary React context providers, or turn expensive loops into web workers if needed). *Week 17:* Security & Hardening – do a pass focusing on security: attempt some known attack patterns (e.g. send unauthorized socket events from a test client) to ensure the server properly rejects them. Add rate limits or debouncing on actions that could be spammed. Ensure all forms/inputs sanitize data (use a library or manual escaping for any user-generated content). *Week 18:* Beta Release – deploy the application to the production

environment. Invite a small group of testers (maybe friends) to play a one-shot game session on the platform. Closely observe for any issues: sync problems, UI confusion, performance hiccups. Gather feedback. *Week 19+:* Iterate on feedback – fix bugs, tweak UX (maybe add a tutorial or help text if testers got confused), and prepare for a wider release. This phased plan ensures core functionality is built first, then refined, with testing and adjustments toward the end.

- **Risks & Mitigations: Scope Creep & Complexity** – This project includes many complex subsystems (mapping, combat, etc.). Mitigation: define clear *MVP features vs optional* ones; implement the must-haves first (basic map, basic combat) before adding extras (fog of war, advanced condition logic). **Performance Bottlenecks** – Rendering a large map with many tokens or running complex vision calculations could lag. Mitigation: use performance-oriented libraries (WebGL via Pixi which is optimized for graphics [10]), implement culling and throttling (don't recalc fog every frame, maybe only on movement stop), and test on mid-range hardware. **Real-time Sync Issues** – Network lag or server outages can disrupt gameplay. Mitigation: use optimistic UI so minor lag is hidden, implement reconnection logic to recover state, and possibly allow manual reconciling (GM can adjust if something goes out of sync). Consider hosting the socket server on a reliable infrastructure and possibly scale-out if many games run (Socket.io can use Redis adapter for multi-server). **Data Consistency** – e.g. two players simultaneously updating inventory could cause duplication or loss. Mitigation: use transactions or row locking in the database for such updates, and have the server arbitrate all changes (clients never directly modify shared state without server ack). **Security & Cheating** – A malicious user might try to call GM-only APIs or modify the client code to give themselves stat points or extra actions. Mitigation: strict server-side checks on all operations (e.g. verify action legitimacy: a player casting a spell must have that spell prepared and resources available; ignore or log any invalid attempts). Use HTTPS/WSS to prevent sniffing or tampering. Possibly use a validation system for critical calculations (like damage) – e.g. re-compute on server to ensure the client didn't fudge dice rolls. **Mobile/Tablet usability** – Some players might use the VTT on an iPad or phone, which could be hard with a complex UI. Mitigation: design responsive layouts (the vertical tab bar could collapse to icons or move to bottom on narrow screens), ensure touch controls for map (enable pinch zoom and drag), and test on touch devices. Provide alternative UI for hover-dependent features (since touch can't hover – e.g. tapping an item could show tooltip via a modal). **Canvas Compatibility** – WebGL can sometimes have issues on older GPUs or certain browsers. Mitigation: have a fallback canvas renderer if Pixi WebGL fails (Pixi does this automatically in many cases). Also, keep an eye on memory leaks in Pixi – use its recommended practices for destroying objects. **Third-Party Library Risks** – relying on many libraries (state mgmt, UI kit, etc.) might introduce bugs or upgrade difficulties. Mitigation: lock versions for stability, and keep each usage relatively isolated (so you can swap out if needed). Choose libraries with strong community support (Redux/RTK or Zustand are well-maintained; Shadcn UI is community-driven but built on Radix which is solid). **Deployment Risks** – a misconfigured server or database could lead to crashes or data loss. Mitigation: use staging to catch deployment issues, and automate tests and backups. Also, start with a small user base to ensure the system scales gradually. By anticipating these risks and addressing them early (with fallback plans), you significantly increase the project's chance of success.

- **Open Questions & Decisions: Game Rules Specifics** – We assume a D&D-like rule set in examples (e.g. advantage, 5-ft grid, HP, mana). But the exact formulas and mechanics need confirmation. Open question: will we use an existing ruleset (like D&D 5e SRD) or a custom simplified system? This affects data (e.g. list of conditions, spells) and algorithms (hit calculation). We proceed with a generic approach that can be tuned to the desired rules. **Map vs Combat Map** – Do we have one unified map for both world exploration and tactical combat, or separate ones? (Our plan assumes the Map tab is for world overview and Fighter tab is a tactical map,

potentially of the same area zoomed in. If separate, we need the ability to load a different map for combat scenes, e.g. a battle grid image). For now, assume the combat uses the same map component but possibly a different tile/background when needed – this is an area to clarify with stakeholders. **Number of Players** – The architecture should handle multiple players in one session (we assumed maybe up to 4-5 plus GM). If a much larger number is needed, we might need to adjust (e.g. many WebSocket connections and more performance optimization). **Turn Order Variations** – Does the system need to support non-standard turn orders or simultaneous turns? Currently planned for classic cyclic turns; if simultaneous turns or real-time mode is needed, that's a different design (not in scope as a turn-based VTT). **State Management Choice** – After evaluation, we decided on **Zustand for global state** (simpler than Redux, with good performance and minimal boilerplate) and **XState for complex finite-state logic** (like managing turn states and skill unlock flows) to reduce bugs [1] [2]. This combo was chosen over Redux Toolkit (which, while powerful and structured, would add unnecessary ceremony for a solo dev project of this scope). **Canvas Library Choice** – We compared using **PixiJS vs Konva** for the map and decided on **PixiJS via React-Pixi** due to its superior performance with many sprites and WebGL acceleration (noting that Foundry VTT successfully uses Pixi for its heavy rendering needs [4]). Konva (canvas2D) is easier for simpler drawings, but for features like dynamic lighting or large maps, Pixi's GPU usage is advantageous. **UI Component Library** – We opted for **Tailwind CSS** for styling and integrated **Shadcn UI** components (built on Radix UI) for accessible, pre-built controls [3]. This decision offers a custom look (since we style with Tailwind) while saving time on complex components (Radix gives us accessible menus, dialogs, etc.). Alternative considered: Material UI (rejected for being too heavy and not matching a fantasy theme out-of-the-box), or Chakra UI (nice and accessible, but we preferred Tailwind's flexibility and the Radix interoperability). **Real-time Transport** – We chose **Socket.io** over raw WebSocket for convenience: it provides automatic reconnection, event namespacing, and room management, which fits our needs (rooms for each game session) [11]. This slightly higher-level approach speeds up development of real-time features. **Backend Framework** – Using **Express with tRPC** (or a lightweight REST) was decided to keep the same language (TypeScript) across stack and benefit from type sharing. We considered GraphQL but determined it unnecessary given our tight coupling of client-server and the real-time nature of many interactions (tRPC's remote procedure call style and built-in WebSocket support suit us well). **Database** – We chose **PostgreSQL** via Prisma for reliable relational storage. Alternative: a NoSQL (like Mongo or Firebase) could simplify real-time sync for some apps, but here we have structured relational data (characters, items with relationships), and transactions (for things like inventory) which Postgres handles well. Prisma gives us type-safe DB access and migration tools, speeding development. We will, however, be careful with Prisma's generated queries for performance (profiling any slow queries). **Content Storage** – If we allow image uploads (for custom tokens or maps), we will likely integrate an S3 bucket for file storage rather than storing binary in Postgres – this detail can be decided when needed (not core to MVP unless custom assets are a goal). All these decisions and their rationales are documented to ensure clarity moving forward and to revisit if requirements change.

Appendix

Glossary of Terms:

- **Virtual Tabletop (VTT):** An online platform that simulates a tabletop RPG experience with maps, tokens, dice rolls, and character sheets. Examples include Roll20 and Foundry VTT.
- **Fog of War (FoW):** A game feature where unexplored or out-of-vision areas of the map are obscured for players. In our VTT, FoW will hide map sections the players' characters have not seen, and dynamically reveal areas as they explore or based on line-of-sight.
- **Line-of-Sight (LoS):** A calculation to determine what a character can see from its position, usually

considering obstacles. LoS is used for revealing fog of war and checking if a target can be seen for a spell/attack. Implemented by casting rays or using geometry to see if any walls block the view between two points.
- **Area of Effect (AoE):** A spell or ability that affects an area (multiple grid cells) rather than a single target. Common AoE shapes are circle (radius), cone, or line. Tokens within an AoE template on the map will all be affected when the spell is cast.
- **Initiative:** A value determining turn order in combat (often a dice roll plus a stat). Higher initiative means acting earlier. We maintain an initiative list each encounter to cycle through turns.
- **Condition:** A status effect on a character or token (e.g. Stunned, Prone, Poisoned). Conditions usually confer some penalty/bonus and have a duration. Our system will track conditions per token with remaining duration (in rounds or seconds).
- **Optimistic UI:** A UX pattern where the interface updates immediately in anticipation of a successful operation, rather than waiting for confirmation. For example, when you move a token, it moves on your screen instantly (optimistically) and then the actual server update confirms it to others – this makes the app feel responsive [8].
- **Conflict Resolution:** In collaborative real-time apps, when two sources of truth conflict (e.g. two players moving the same token), the system must resolve it. We plan simple rules: server is authoritative and last action wins, with the GM's actions taking precedence in case of conflict.
- **Redux / Zustand / XState:** Different state management tools. Redux is a traditional centralized store (with Toolkit simplifying syntax), Zustand is a lightweight hook-based state store (less boilerplate), and XState manages finite state machines (useful for game flow logic). We use a mix: Zustand for global app state and XState for specific game logic sequences.
- **React-Pixi / PixiJS:** A rendering library for 2D graphics using WebGL. It treats game elements as sprites and can render thousands of objects efficiently with GPU acceleration. We use React-Pixi to integrate Pixi with React components.
- **React Flow:** A library for building node/link diagrams in React with pan and zoom – perfect for skill trees or flow charts [7]. It handles dragging, connecting nodes, etc., though our skill tree will be static structure with predetermined connections.
- **Socket.io:** A popular library for real-time bi-directional communication. It abstracts WebSocket and provides features like automatic reconnection, ping/pong heartbeats, and the concept of rooms for grouping connections – which we leverage for game sessions.
- **Prisma:** An ORM (Object-Relational Mapper) for Node/TS that simplifies database access by generating a type-safe API for queries. It will let us define models like Character, Item, etc., and handle migrations to PostgreSQL.
- **Audit Log:** A record of significant actions or events, mainly for debugging or security. Our AuditLog will store events like "Player X gave themselves 5 stat points" or "GM Y spawned a dragon token" with timestamps, so we have an immutable history for dispute resolution or error tracking.

Sample Data Formats (JSON as Tables):

- **Character Object Example:** A representation of a player character's data.

Field	Sample Value	Description
id	`"char_12345"`	Unique character ID (string or UUID).
name	`"Aria Strongbow"`	Character's name.
level	5	Current level.
class	`"Ranger"`	Class or archetype.

Field	Sample Value	Description
baseStats	STR: 8, DEX: 14, INT: 10, etc	Base attributes (could be an object with each stat).
derivedStats	HP: 45, AC: 16, Attack: +5	Derived stats calculated from base + gear (could store or compute on the fly).
statPointsUnused	2	Unallocated stat points available.
gearSlots	`{ head: item_1001, weapon: item_1005, ... }`	Mapping of equipment slots to Item IDs (null if empty).
inventory	`[ item_1002, item_1003, ... ]`	Array of Item IDs in backpack.
spellsKnown	`[ spell_fireball, spell_heal, ... ]`	IDs of spells the character has learned.
spellsPrepared	`[ spell_fireball, spell_cure_wounds ]`	Subset of spells currently prepared (if applicable).
skillNodesUnlocked	`[ node_a1, node_b3 ]`	IDs of skill tree nodes unlocked by this character.
conditions	`[ {type:"Poisoned", turnsRemaining:1} ]`	Active conditions affecting the character.
roomId	`"room_98765"`	The game room this character is in (for quick lookup).
userId	`"user_555"`	Reference to the User who owns this character.

- **Item Object Example:** Represents an item (for inventory or equipment).

Field	Sample Value	Description
id	`"item_1001"`	Unique item ID.
name	`"Iron Longsword"`	Item name.
type	`"Weapon"`	Category/type (Weapon, Armor, Potion, etc.).
rarity	`"Common"`	Rarity tier (Common/Uncommon/ Rare/Epic).
slot	`"weapon"`	Applicable equipment slot (if equippable).
statModifiers	`{ Attack: +2, Damage: "1d8" }`	Effects on stats (Attack +2, adds 1d8 damage roll).
requirements	`{ level: 3 }`	Requirements to use (e.g. must be level 3).

Field	Sample Value	Description
quantity	1	Quantity (stackable items).
description	`"A sturdy iron sword favored by guards."`	Text description or lore.

- **Spell Object Example:** Represents a spell's static definition.

Field	Sample Value	Description
id	`"spell_fireball"`	Unique spell identifier.
name	`"Fireball"`	Spell name.
level	3	Spell level or tier.
school	`"Evocation"`	Magical school or category.
range	150	Range in feet (or units).
area	`20` (feet radius)	Area of effect (if any, here 20-ft radius).
target	`"Area (sphere)"`	Target type (e.g. Single, Area, Self).
cost	`5 MP`	Resource cost (mana points, etc.).
cooldown	`null`	Cooldown duration (null if none, or e.g. 2 rounds).
effect	`"8d6 fire damage"`	Effect description (damage, healing, etc.).
save	`"DEX DC 14 for half"`	Saving throw info (if applicable).
conditionsInflicted	`["Burning"]`	Condition applied by spell (if any).
description	`"A bright streak flashes... (full text)"`	Full spell description.

- **Skill Tree Data Examples:**

- *SkillNode:*

Field	Sample	Description
id	`"node_a1"`	Node ID.
name	`"Deadeye"`	Name of the passive skill.
description	`"+5% Critical Hit Chance"`	Effect of the skill.
position	`{x:1, y:3}`	Grid position in layout (for rendering).

Field	Sample	Description
prerequisites	`[ "node_a0" ]`	IDs of prerequisite nodes (must unlock first).
unlockedBy	`null`	Character ID if this is stored per character unlock (or this field is not in static definition, rather stored in character as above).

- *SkillEdge:* (could also be implicit from prereqs)

Field	Value	Description
fromNode	`"node_a0"`	Edge from node A0 …
toNode	`"node_a1"`	… to node A1 (meaning A1 depends on A0).

- **Encounter Example:** Represents an active combat encounter.

Field	Sample Value	Description
id	`"enc_001122"`	Encounter ID.
roomId	`"room_98765"`	The game room where encounter takes place.
mapId	`"map_forest01"`	The ID of the map or scene used.
round	2	Current round number.
turnIndex	4	Index in initiative order of current turn.
initiativeOrder	`[ "char_12345", "npc_goblin01", "char_67890", ...]`	Ordered list of participant IDs.
participants	Array of participant objects (see below)	Combatants involved and their state.
status	`"active"`	Encounter status (active or ended/paused).
createdBy	`"user_GM001"`	Who started the encounter (GM's user ID).
startTime	timestamp	When encounter began (for logs).

- *Participant (within encounter participants):*

Field	Sample	Description
id	`"npc_goblin01"`	Token/character ID of participant.
type	`"NPC"`	Type (PC or NPC/monster).

Field	Sample	Description
name	`"Goblin"`	Name (for display).
initiative	15	Initiative roll result.
HP	5	Current hit points (if it drops to 0, remove or mark defeated).
conditions	`[ {type:"Stunned", turnsRemaining:1} ]`	Active conditions on this participant.
isActive	false	Whether it's currently this participant's turn.

• **Real-time Event Payload Examples:** (as tables)

• *Token Movement Event (`token.move`)*

Field	Example	Description
event	`"token.move"`	Event name/type.
roomId	`"room_98765"`	Room in which this event is happening.
tokenId	`"char_12345"`	ID of token being moved.
from	`{x: 10, y: 5}`	Previous grid coordinates (optional).
to	`{x: 14, y: 5}`	New grid coordinates of the token.
remainingSpeed	0	Movement left this turn (feet or tiles).
by	`"user_555"`	Who initiated the move (user ID).
timestamp	1633024800	Timestamp of event (epoch).

• *Spell Cast Event (`action.castSpell`)*

Field	Example	Description
event	`"action.castSpell"`	Event type for casting a spell action.
casterId	`"char_12345"`	ID of the character casting the spell.
spellId	`"spell_fireball"`	ID of the spell being cast.
targets	`[ "npc_goblin01", "npc_goblin02" ]`	IDs of targets affected.
result	`{ "npc_goblin01": {"damage": 18, "status": "alive"}, "npc_goblin02": {"damage": 18, "status": "dead"} }`	Outcome per target (damage dealt, and if they died).

Field	Example	Description
friendlyFire	false	Indicator if allies were hit (if applicable).
by	`"user_555"`	Who performed (should match caster's user).
timestamp	1633024820	Timestamp of cast event.

- *Condition Applied Event (`token.conditionApplied`)*

Field	Example	Description
event	`"token.conditionApplied"`	Event for adding a condition to a token.
tokenId	`"npc_goblin01"`	Target token receiving the condition.
condition	`"Stunned"`	Type of condition applied.
duration	1	Duration in turns (or rounds).
source	`"char_12345"`	Who caused it (e.g. caster ID).
timestamp	1633024825	Time of application.

(Note: All events would be broadcast by the server to all clients in the room. Some events might be collapsed or combined for efficiency – e.g. an `encounter.start` event might include the full turn order and initial stats of all participants to initialize clients.)

Recommended Reading / Resources:

- **React Router –** *Guidance on implementing nested routes and layouts in React Router v6.* (See React Router official documentation on Layout Routes for structuring the tabbed UI.) [12]
- **React Flow Documentation –** *Learn how to use React Flow for building node-based UIs (skill trees).* [7]
- **XState + React Guide –** *Using XState state machines in React apps for complex UI logic (e.g. turn management). It covers state modeling and integration.* [13] [2]
- **Pixi.js Official Guide –** *Basics of PixiJS for high-performance 2D rendering.* (Refer to the PixiJS documentation for displaying images, containers, and optimizing performance; see also Foundry VTT's notes on using Pixi [4].)
- **Socket.io Documentation –** *Setup and usage of Socket.io for real-time features.* (Includes sections on rooms, namespaces, and best practices for error handling and reconnection which are relevant to this project's real-time system.) [11]
- **Shadcn UI + Radix –** *Overview of Shadcn UI components built on Radix and Tailwind.* (This resource explains how Shadcn provides accessible pre-built components that we use to speed up UI development) [3].
- **Accessibility in Games –** *WCAG 2.1 Primer for Game Interfaces.* (General guide on ensuring UI elements like color usage, text size, and interaction feedback meet accessibility standards, which is crucial for our diverse user base.)

Definition of Done Checklist:

- [ ] **Navigation & Layout:** All main tabs (Map, Character, Spells, Skill Tree, Fighter) are implemented with a consistent layout. Left tab bar is functional (clicking switches content) and accessible (keyboard and screen-reader). Responsive design verified on desktop and tablet screens.
- [ ] **Map Functionality:** Map tab supports smooth pan/zoom of a large image or tile map. Markers appear appropriately and cluster or hide at zoomed-out levels. Optional overlays (grid, fog of war, measuring tool) toggle correctly without errors. Users can interact with markers (e.g. click for info) and the map performance meets FPS target.
- [ ] **Character Management:** Character tab displays all relevant info (stats, gear, inventory) correctly from server data. Stat allocation flow works with validation (cannot overspend points) and updates both UI and backend. Equipping and unequipping items via drag-drop or buttons updates the UI (slots/inventory) and character stats in real-time (and persists to server). Tooltips and other UI affordances (e.g. item rarity highlighting) are implemented.
- [ ] **Spells & Abilities:** Spells tab lists the character's spells (or all spells if design) and can be filtered/sorted. The player can select spells into a "prepared" list up to the allowed number, and the selection state is saved. Spell detail view shows comprehensive info. Any cooldown or resource indicator updates after spell use (if applicable).
- [ ] **Skill Tree:** The skill tree graph is rendered and interactive. Pan/zoom works without visual glitches. All skill nodes are present and connected by lines. Unlocking a skill by spending a point immediately reflects in UI (node changes state) and deducts the point (and persists). Prerequisite logic prevents illegal unlocks. Respec functionality (if provided) correctly refunds points and locks appropriate nodes.
- [ ] **Combat System:** Fighter tab allows the GM to start an encounter and all players see the initiative turn order. Turn logic proceeds correctly on "End Turn" clicks, cycling through combatants and rounds. Movement is restricted by speed (the system prevents or corrects moves that are too far). Attack and spell actions can be executed: targets can be selected, and the system applies damage and conditions appropriately (with server verification). AoE templates correctly identify affected tokens. Condition icons/statuses update on tokens and expire after the set duration. The GM's special controls (spawn token, adjust HP, pause combat) all function and broadcast to players. Combat can be cleanly ended/reset.
- [ ] **Real-Time Sync:** All game-relevant actions are synchronized across clients in a room. Test with two clients: moving a token on one reflects on the other within a negligible delay; stat changes, item equips, spell prepares, skill unlocks, and combat actions all propagate and remain consistent. The system handles a client refresh/reconnect smoothly (on rejoining, the client state is up-to-date). No duplicated or out-of-order events cause inconsistent state (verified via testing scenarios).
- [ ] **State Management & Performance:** The app's state updates are efficient – no noticeable lag on UI updates when state changes (e.g. equipping item recalculating stats). No memory leaks or runaway CPU usage in long sessions (tested by an hour of usage or automated soak test). The bundle size is within acceptable limits (split by routes so initial load is fast). Frame rate is good even with ~20 tokens on map and moderate animations.
- [ ] **Security & Roles:** All critical actions are protected by proper permission checks (verified by attempting disallowed actions as a player – e.g. a player cannot spawn a token or modify another player's stats). Input fields and communications are secure: no XSS (tested by inputting script tags in text fields), no SQL injection (Prisma queries parameterized), and sensitive data (like passwords or tokens) never exposed in client or logs. Basic rate limiting is in place (e.g. one can't spam join attempts or flood the server with move events excessively without throttle).
- [ ] **Testing & QA:** Unit tests cover game logic (calculations, reducers) with high pass rate. Integration/UI tests cover key user flows (character creation or load, performing an attack, etc.)

and all pass in CI. All high-severity bugs from testing have been fixed. The app has been tested on at least two browsers (e.g. Chrome, Firefox) for compatibility.

- [ ] **Documentation & Deployment:** README or developer docs exist for setting up the project, running the server, and deploying. The architecture diagram and decision log are updated to reflect the final implementation. The system is deployed to a production environment, and a backup/monitoring system is active. We have run a final smoke test on production to ensure everything (auth, each tab, combat, sync) works with real deployment settings.

Once all checkboxes are satisfied, the RPG Virtual Tabletop is feature-complete, polished, and ready for players and GMs to enjoy their adventures online. [4] [1]

---

[1] Expertivia - Mastering State Management in React: A 2024 Comparison (Redux Toolkit vs. Zustand vs. Jotai)
https://www.expertivia.net/en/articles/mastering-state-management-in-react-a-2024-comparison-redux-toolkit-vs-zustand-vs-jotai-6yxuqvbnmd

[2] [9] [13] Thoughts on Building a Game with XState | Asuka Wang
https://asukawang.com/blog/thoughts-on-building-a-game-with-xstate/

[3] My Tiny Guide to Shadcn, Radix, and Tailwind | by Mairaj Pirzada | Medium
https://medium.com/@immairaj/my-tiny-guide-to-shadcn-radix-and-tailwind-da50fce3140a

[4] [11] Show HN: I built a virtual tabletop for playing Dungeons and Dragons | Hacker News
https://news.ycombinator.com/item?id=37857765

[5] [6] Character Screen — Shakes & Fidget Help Center
https://playa-games.helpshift.com/hc/en/4-shakes-fidget-1653988985/faq/107-character-screen/

[7] [12] Any good react libraries for skill trees or tech trees? : r/react
https://www.reddit.com/r/react/comments/189zlap/any_good_react_libraries_for_skill_trees_or_tech/

[8] Optimistic UI: Making Apps Feel Faster (Even When They're Not)
https://medium.com/@alexglushenkov/optimistic-ui-making-apps-feel-faster-even-when-theyre-not-ea296bc84720

[10] graphics - Pixi.js vs Konva.js vs D3.js - Stack Overflow
https://stackoverflow.com/questions/57948360/pixi-js-vs-konva-js-vs-d3-js