Design Decisions – Token Trading Table

This document explains the key architectural, technical, and UI/UX decisions made while building the **Token Trading Table (Axiom Trade – Pulse replica)**. The goal was to achieve **pixel-perfect UI**, **high performance**, and **clean, reusable architecture** while strictly following the given requirements.

---

1. Overall Architecture

Atomic Design Methodology

The project follows **Atomic Architecture** to ensure scalability, reusability, and separation of concerns.

Structure: - **atoms/** – Smallest UI units (icons, buttons, badges, skeleton blocks, tooltips) - **molecules/** – Combinations of atoms (search bar, token card, modal content, popovers) - **organisms/** – Complex UI sections (TokenTable, TokenRow, Header, Tabs)

Why this approach? - Promotes reusability across the app - Easier to maintain and test - Prevents duplication and UI inconsistency

---

2. Tech Stack Decisions

Next.js 14 (App Router)

- Chosen for **server-first rendering**, better performance, and layout-level optimizations
- App Router enables colocated layouts, loading states, and error boundaries

TypeScript (Strict Mode)

- Enforced strict typing for:
- Token models
- Sorting keys
- Redux state
- WebSocket payloads
- Prevents runtime bugs and improves refactor confidence

Tailwind CSS

- Enables **pixel-perfect precision** (exact spacing, font sizes, colors)
- Utility-first approach avoids CSS bloat
- Matches Figma/production UI down to ≤2px difference

---

3. State Management Strategy

Redux Toolkit (Global & Complex State)

Used for: - Token lists per column (New Pairs, Final Stretch, Migrated) - Sorting state (key + order per column) - UI states shared across components (selected tab, filters)

Why Redux Toolkit? - Predictable state updates - DevTools for debugging - Clean slices (`tokenSlice.ts`) with minimal boilerplate

React Query (Server / Async State)

Used for: - Token data fetching - Refetching & caching - Loading, error, and retry handling

Benefit: - Avoids duplicating async state logic in Redux - Keeps Redux focused on UI & interaction state

---

4. Real-Time Price Updates

WebSocket Mock Hook

Implemented via `useWebSocketMock.ts`: - Simulates real-time price updates - Emits random price changes at intervals - Mimics real exchange behavior without backend dependency

Price Flash Logic

Handled by `usePriceFlash.ts`: - Detects price increase / decrease - Applies temporary green/red flash - Smooth transitions using CSS and `requestAnimationFrame`

Design goal: - Zero layout shift - No unnecessary re-renders

---

5. Performance Optimizations

Memoization

- `React.memo` on TokenRow, TokenTable, and heavy components
- `useMemo` for sorted & filtered token lists
- `useCallback` for event handlers

Rendering Strategy

- Row-level updates only (price cell updates, not full table)
- Stable keys and fixed row heights to prevent reflow

Interaction Performance

- Hover effects via CSS only (no JS)
- Click interactions < 100ms
- No blocking renders during WebSocket updates

---

6. UI Interactions & Accessibility

Interaction Variety

- **Tooltip** – quick info on hover
- **Popover** – contextual token actions
- **Modal** – detailed token view
- **Sorting** – column-based sorting with visual indicators

Accessibility

- Radix UI / Headless UI components
- Keyboard navigable popovers & modals
- Proper ARIA attributes

---

7. Loading & Error Handling

Loading States

- Skeleton loaders for table rows
- Shimmer effects for perceived performance
- Progressive loading for large lists

Error Boundaries

- Page-level and component-level error boundaries
- Graceful fallback UI with retry actions

---

8. Pixel-Perfect UI Strategy

Visual Accuracy

- Matched font sizes, weights, spacing, and colors
- Compared against live site using visual regression tools
- Ensured ≤2px deviation

Responsive Design

- Fully responsive down to **320px width**
- Horizontal scroll handling for small screens
- Touch-friendly interactions on mobile

---

9. Code Quality & Maintainability

- Centralized types (`types/`, `interface.ts`)
- Shared utilities (`utils/format`, `helpers.ts`)
- Toast & notification abstraction
- Clear folder boundaries and naming conventions

Complex logic is documented inline for easier onboarding.

---

10. Lighthouse & Quality Targets

- Achieved **≥90 Lighthouse score** (Mobile & Desktop)
- No CLS (Cumulative Layout Shift)
- Optimized images & fonts
- Minimal JS execution during interactions

---

11. Trade-offs & Future Improvements

Trade-offs

- WebSocket is mocked (as backend not provided)
- Charts simplified to focus on table performance

Future Enhancements

- Real WebSocket integration
- Virtualized rows for very large datasets
- Snapshot-based visual regression CI

Conclusion

This project prioritizes **performance, precision, and reusability**. Every component is designed to be reusable, accessible, and optimized for real-time trading use cases while maintaining a pixel-perfect UI.
