Historical Trade

**Kurzora Historical Trades Page - Complete UI Analysis**

**13-Point Framework for Immediate Cursor Implementation**

**1. UI Components & Layout**

**Interactive Elements**

**Primary Components:**

* **BackButton** (navigation to dashboard)
* **SummaryMetricCards** (Total Trades, Total P&L, Win Rate)
* **TradeHistoryTable** (main data display with sorting)
* **DateRangeFilter** (filter trades by date period)
* **StatusFilter** (filter by close reason)
* **ExportButton** (download trades as CSV/PDF)
* **TradeDetailModal** (detailed view on row click)
* **PaginationControls** (for large datasets)

**Loading & Error States:**

* **HistoryTableSkeleton** (loading placeholders)
* **EmptyHistoryState** (no trades message)
* **ErrorBoundary** (fallback for data errors)

**React + TypeScript Component Structure**

// Complete Historical Trades Architecture

<Layout>

<div className="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8 py-8">

{/\* Header Section \*/}

<div className="mb-8">

<div className="flex items-center justify-between mb-6">

<div className="flex items-center space-x-4">

<BackButton onClick={handleBackToDashboard} />

<h1 className="text-3xl font-bold text-white">Historical Trades</h1>

</div>

<div className="flex items-center space-x-4">

<DateRangeFilter onFilterChange={handleDateFilter} />

<StatusFilter onFilterChange={handleStatusFilter} />

<ExportButton onExport={handleExport} />

</div>

</div>

{/\* Summary Metrics \*/}

<div className="grid grid-cols-1 md:grid-cols-4 gap-6 mb-8">

<SummaryMetricCard

icon={BarChart3}

title="Total Trades"

value={totalTrades}

color="blue"

/>

<SummaryMetricCard

icon={totalPnL >= 0 ? TrendingUp : TrendingDown}

title="Total P&L"

value={`${totalPnL >= 0 ? '+' : ''}$${totalPnL.toFixed(0)}`}

color={totalPnL >= 0 ? 'emerald' : 'red'}

/>

<SummaryMetricCard

icon={TrendingUp}

title="Win Rate"

value={`${winRate.toFixed(1)}%`}

color="amber"

/>

<SummaryMetricCard

icon={Calculator}

title="Avg. Return"

value={`${avgReturn.toFixed(1)}%`}

color="purple"

/>

</div>

</div>

{/\* Main Table \*/}

<Card className="bg-slate-800/50 backdrop-blur-sm border-slate-700">

<CardHeader>

<CardTitle className="text-lg text-white flex items-center space-x-2">

<BarChart3 className="h-5 w-5 text-blue-400" />

<span>📊 Trading Performance Analysis</span>

</CardTitle>

</CardHeader>

<CardContent>

{loading ? (

<HistoryTableSkeleton />

) : filteredTrades.length > 0 ? (

<>

<TradeHistoryTable

trades={paginatedTrades}

onRowClick={handleTradeDetailView}

sortConfig={sortConfig}

onSort={handleSort}

/>

<PaginationControls

currentPage={currentPage}

totalPages={totalPages}

onPageChange={setCurrentPage}

/>

</>

) : (

<EmptyHistoryState />

)}

</CardContent>

</Card>

{/\* Trade Detail Modal \*/}

{selectedTrade && (

<TradeDetailModal

trade={selectedTrade}

isOpen={isDetailModalOpen}

onClose={closeDetailModal}

/>

)}

{/\* Paper Trading Disclaimer \*/}

<div className="text-xs text-gray-500 text-center mt-4">

\*These are simulated paper trading results. No real capital was involved.

</div>

</div>

</Layout>

**Responsive Design Considerations**

// Mobile-first responsive table

const ResponsiveTradeTable = () => {

const [isMobile] = useMediaQuery('(max-width: 768px)');

return isMobile ? (

<div className="space-y-4">

{trades.map(trade => (

<TradeCard key={trade.id} trade={trade} />

))}

</div>

) : (

<Table>

{/\* Desktop table layout \*/}

</Table>

);

};

**2. State Management (Zustand)**

**Historical Trades Store Structure**

interface HistoricalTradesStore {

// Data state

trades: ClosedTrade[];

filteredTrades: ClosedTrade[];

selectedTrade: ClosedTrade | null;

// UI state

loading: boolean;

error: string | null;

currentPage: number;

itemsPerPage: number;

sortConfig: SortConfig;

// Filter state

filters: {

dateRange: { start: Date; end: Date };

status: string[];

minAmount: number;

maxAmount: number;

symbols: string[];

};

// Computed values

totalTrades: number;

totalPnL: number;

winRate: number;

avgReturn: number;

bestTrade: ClosedTrade | null;

worstTrade: ClosedTrade | null;

// Actions

loadTrades: () => Promise<void>;

updateFilters: (filters: Partial<TradeFilters>) => void;

resetFilters: () => void;

sortTrades: (column: string, direction: 'asc' | 'desc') => void;

setCurrentPage: (page: number) => void;

selectTrade: (trade: ClosedTrade) => void;

exportTrades: (format: 'csv' | 'pdf') => Promise<void>;

// Real-time updates

subscribeToUpdates: () => void;

unsubscribeFromUpdates: () => void;

}

export const useHistoricalTradesStore = create<HistoricalTradesStore>((set, get) => ({

// Initial state

trades: [],

filteredTrades: [],

selectedTrade: null,

loading: false,

error: null,

currentPage: 1,

itemsPerPage: 10,

sortConfig: { column: 'closedDate', direction: 'desc' },

filters: {

dateRange: {

start: subMonths(new Date(), 3),

end: new Date()

},

status: [],

minAmount: 0,

maxAmount: Infinity,

symbols: []

},

// Computed getters

get totalTrades() { return get().filteredTrades.length; },

get totalPnL() {

return get().filteredTrades.reduce((sum, trade) => sum + trade.pnl, 0);

},

get winRate() {

const trades = get().filteredTrades;

const winningTrades = trades.filter(t => t.pnl > 0).length;

return trades.length > 0 ? (winningTrades / trades.length) \* 100 : 0;

},

get avgReturn() {

const trades = get().filteredTrades;

const totalReturn = trades.reduce((sum, trade) => sum + trade.pnlPercent, 0);

return trades.length > 0 ? totalReturn / trades.length : 0;

},

// Actions

loadTrades: async () => {

set({ loading: true, error: null });

try {

const response = await fetch('/api/trades/history', {

headers: { 'Authorization': `Bearer ${getAuthToken()}` }

});

const data = await response.json();

set({

trades: data.trades,

loading: false,

lastUpdated: new Date().toISOString()

});

get().applyFilters();

} catch (error) {

set({ error: error.message, loading: false });

}

},

updateFilters: (newFilters) => {

set(state => ({

filters: { ...state.filters, ...newFilters },

currentPage: 1 // Reset to first page

}));

get().applyFilters();

},

applyFilters: () => {

const { trades, filters, sortConfig } = get();

let filtered = trades.filter(trade => {

const tradeDate = new Date(trade.closedDate);

const inDateRange = tradeDate >= filters.dateRange.start &&

tradeDate <= filters.dateRange.end;

const matchesStatus = filters.status.length === 0 ||

filters.status.includes(trade.reasonForClosing);

const inAmountRange = Math.abs(trade.pnl) >= filters.minAmount &&

Math.abs(trade.pnl) <= filters.maxAmount;

const matchesSymbol = filters.symbols.length === 0 ||

filters.symbols.includes(trade.symbol);

return inDateRange && matchesStatus && inAmountRange && matchesSymbol;

});

// Apply sorting

filtered.sort((a, b) => {

const aValue = a[sortConfig.column];

const bValue = b[sortConfig.column];

const direction = sortConfig.direction === 'asc' ? 1 : -1;

if (typeof aValue === 'number') {

return (aValue - bValue) \* direction;

}

return aValue.localeCompare(bValue) \* direction;

});

set({ filteredTrades: filtered });

},

exportTrades: async (format) => {

const { filteredTrades } = get();

const response = await fetch('/api/trades/export', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${getAuthToken()}`

},

body: JSON.stringify({ trades: filteredTrades, format })

});

const blob = await response.blob();

const url = window.URL.createObjectURL(blob);

const a = document.createElement('a');

a.href = url;

a.download = `trades-history-${new Date().toISOString().split('T')[0]}.${format}`;

a.click();

}

}));

**3. API Contracts & Integration**

**API Endpoints**

// GET /api/trades/history - Fetch trade history

interface TradeHistoryRequest {

startDate?: string;

endDate?: string;

status?: string[];

symbols?: string[];

page?: number;

limit?: number;

sortBy?: string;

sortOrder?: 'asc' | 'desc';

}

interface TradeHistoryResponse {

trades: ClosedTrade[];

pagination: {

currentPage: number;

totalPages: number;

totalItems: number;

itemsPerPage: number;

};

summary: {

totalPnL: number;

winRate: number;

avgReturn: number;

bestTrade: ClosedTrade;

worstTrade: ClosedTrade;

};

lastUpdated: string;

}

// POST /api/trades/export - Export trades

interface ExportRequest {

trades: ClosedTrade[];

format: 'csv' | 'pdf';

includeCharts?: boolean;

}

// GET /api/trades/analytics - Advanced analytics

interface AnalyticsResponse {

monthlyPerformance: MonthlyStats[];

sectorPerformance: SectorStats[];

timeframeAnalysis: TimeframeStats[];

riskMetrics: {

sharpeRatio: number;

maxDrawdown: number;

volatility: number;

};

}

**TypeScript Interfaces**

interface ClosedTrade {

id: string;

userId: string;

symbol: string;

name: string;

sector: string;

// Trade execution

entryPrice: number;

exitPrice: number;

shares: number;

investmentAmount: number;

// Performance metrics

pnl: number;

pnlPercent: number;

score: number;

// Trade management

stopLoss?: number;

takeProfit?: number;

reasonForClosing: 'Target Hit' | 'Stop Loss' | 'Manual Exit' | 'Reversal' | 'Time Decay';

// Timestamps

openedAt: string;

closedAt: string;

closedDate: string; // formatted date

holdingPeriod: number; // in days

// Analysis data

signalData?: {

originalScore: number;

indicators: TechnicalIndicators;

market\_conditions: MarketConditions;

};

}

interface TradeFilters {

dateRange: { start: Date; end: Date };

status: string[];

minAmount: number;

maxAmount: number;

symbols: string[];

sectors: string[];

minScore: number;

maxScore: number;

}

interface SortConfig {

column: keyof ClosedTrade;

direction: 'asc' | 'desc';

}

**4. Performance & Optimization**

**Lazy Loading Implementation**

// Component lazy loading

const TradeDetailModal = lazy(() => import('../components/TradeDetailModal'));

const ExportDialog = lazy(() => import('../components/ExportDialog'));

const AdvancedFilters = lazy(() => import('../components/AdvancedFilters'));

// Table virtualization for large datasets

import { FixedSizeList as List } from 'react-window';

const VirtualizedTradeTable = ({ trades, itemHeight = 60 }) => {

const Row = ({ index, style }) => (

<div style={style}>

<TradeRow trade={trades[index]} />

</div>

);

return (

<List

height={600}

itemCount={trades.length}

itemSize={itemHeight}

overscanCount={5}

>

{Row}

</List>

);

};

**Memoization Strategies**

// Expensive calculations

const TradeSummary = React.memo(({ trades }) => {

const metrics = useMemo(() => ({

totalPnL: trades.reduce((sum, t) => sum + t.pnl, 0),

winRate: (trades.filter(t => t.pnl > 0).length / trades.length) \* 100,

avgReturn: trades.reduce((sum, t) => sum + t.pnlPercent, 0) / trades.length,

bestTrade: trades.reduce((best, current) =>

current.pnl > best.pnl ? current : best, trades[0]

)

}), [trades]);

return <SummaryCards metrics={metrics} />;

});

// Table row memoization

const TradeRow = React.memo(({ trade, onSelect }) => {

const handleClick = useCallback(() => onSelect(trade), [trade, onSelect]);

return (

<TableRow onClick={handleClick} className="hover:bg-slate-700/30">

{/\* Row content \*/}

</TableRow>

);

});

**Caching Strategy**

// React Query for server state management

const useTradeHistory = (filters: TradeFilters) => {

return useQuery({

queryKey: ['tradeHistory', filters],

queryFn: () => fetchTradeHistory(filters),

staleTime: 5 \* 60 \* 1000, // 5 minutes

cacheTime: 30 \* 60 \* 1000, // 30 minutes

refetchOnWindowFocus: false,

keepPreviousData: true

});

};

// Local storage for filter preferences

const useFilterPreferences = () => {

const [filters, setFilters] = useLocalStorage('trade-filters', defaultFilters);

const updateFilters = useCallback((newFilters) => {

setFilters(prev => ({ ...prev, ...newFilters }));

}, [setFilters]);

return [filters, updateFilters];

};

**5. Database Schema**

**PostgreSQL Tables**

-- Closed trades table (main)

CREATE TABLE closed\_trades (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

user\_id UUID NOT NULL REFERENCES users(id),

original\_signal\_id UUID REFERENCES trading\_signals(id),

-- Basic trade info

symbol VARCHAR(10) NOT NULL,

company\_name VARCHAR(255) NOT NULL,

sector VARCHAR(50),

market VARCHAR(20) DEFAULT 'US',

-- Trade execution

entry\_price DECIMAL(10,2) NOT NULL,

exit\_price DECIMAL(10,2) NOT NULL,

shares INTEGER NOT NULL,

investment\_amount DECIMAL(12,2) NOT NULL,

-- Performance

realized\_pnl DECIMAL(12,2) NOT NULL,

pnl\_percentage DECIMAL(5,2) NOT NULL,

original\_signal\_score INTEGER CHECK (original\_signal\_score >= 0 AND original\_signal\_score <= 100),

-- Trade management

stop\_loss\_price DECIMAL(10,2),

take\_profit\_price DECIMAL(10,2),

reason\_for\_closing VARCHAR(50) NOT NULL

CHECK (reason\_for\_closing IN ('Target Hit', 'Stop Loss', 'Manual Exit', 'Reversal', 'Time Decay')),

-- Timestamps

opened\_at TIMESTAMP WITH TIME ZONE NOT NULL,

closed\_at TIMESTAMP WITH TIME ZONE NOT NULL,

holding\_period\_days INTEGER GENERATED ALWAYS AS (

EXTRACT(epoch FROM (closed\_at - opened\_at)) / 86400

) STORED,

-- Analysis data (JSONB for flexibility)

signal\_metadata JSONB,

market\_conditions JSONB,

exit\_analysis JSONB,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Performance optimization indexes

CREATE INDEX idx\_closed\_trades\_user\_date ON closed\_trades(user\_id, closed\_at DESC);

CREATE INDEX idx\_closed\_trades\_symbol ON closed\_trades(symbol);

CREATE INDEX idx\_closed\_trades\_sector ON closed\_trades(sector);

CREATE INDEX idx\_closed\_trades\_pnl ON closed\_trades(realized\_pnl);

CREATE INDEX idx\_closed\_trades\_reason ON closed\_trades(reason\_for\_closing);

CREATE INDEX idx\_closed\_trades\_score ON closed\_trades(original\_signal\_score);

-- Composite index for common queries

CREATE INDEX idx\_closed\_trades\_user\_filters ON closed\_trades(

user\_id, closed\_at, reason\_for\_closing, original\_signal\_score

);

-- Trade analytics materialized view for faster calculations

CREATE MATERIALIZED VIEW user\_trade\_analytics AS

SELECT

user\_id,

COUNT(\*) as total\_trades,

SUM(realized\_pnl) as total\_pnl,

AVG(realized\_pnl) as avg\_pnl,

AVG(pnl\_percentage) as avg\_return,

ROUND((COUNT(\*) FILTER (WHERE realized\_pnl > 0)::DECIMAL / COUNT(\*)) \* 100, 2) as win\_rate,

MAX(realized\_pnl) as best\_trade\_pnl,

MIN(realized\_pnl) as worst\_trade\_pnl,

AVG(holding\_period\_days) as avg\_holding\_period,

DATE\_TRUNC('month', closed\_at) as month\_year,

sector,

COUNT(\*) as trades\_per\_sector

FROM closed\_trades

WHERE closed\_at >= NOW() - INTERVAL '2 years'

GROUP BY user\_id, DATE\_TRUNC('month', closed\_at), sector;

-- Refresh analytics view daily

CREATE OR REPLACE FUNCTION refresh\_trade\_analytics()

RETURNS void AS $$

BEGIN

REFRESH MATERIALIZED VIEW CONCURRENTLY user\_trade\_analytics;

END;

$$ LANGUAGE plpgsql;

-- Monthly performance aggregation table

CREATE TABLE monthly\_performance (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

user\_id UUID NOT NULL REFERENCES users(id),

month\_year DATE NOT NULL,

total\_trades INTEGER NOT NULL DEFAULT 0,

total\_pnl DECIMAL(12,2) NOT NULL DEFAULT 0,

win\_rate DECIMAL(5,2) NOT NULL DEFAULT 0,

avg\_return DECIMAL(5,2) NOT NULL DEFAULT 0,

best\_trade\_pnl DECIMAL(12,2),

worst\_trade\_pnl DECIMAL(12,2),

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

UNIQUE(user\_id, month\_year)

);

**6. User Experience**

**Loading States & Skeleton Screens**

const HistoryTableSkeleton = () => (

<div className="space-y-4">

{/\* Summary cards skeleton \*/}

<div className="grid grid-cols-1 md:grid-cols-4 gap-6 mb-8">

{[...Array(4)].map((\_, i) => (

<Card key={i} className="bg-slate-800/50">

<CardContent className="p-6">

<div className="animate-pulse">

<div className="h-4 bg-slate-700 rounded w-20 mb-2"></div>

<div className="h-6 bg-slate-700 rounded w-16"></div>

</div>

</CardContent>

</Card>

))}

</div>

{/\* Table skeleton \*/}

<div className="space-y-3">

{[...Array(8)].map((\_, i) => (

<div key={i} className="animate-pulse flex space-x-4 p-4">

<div className="h-4 bg-slate-700 rounded w-16"></div>

<div className="h-4 bg-slate-700 rounded w-20"></div>

<div className="h-4 bg-slate-700 rounded w-20"></div>

<div className="h-4 bg-slate-700 rounded w-12"></div>

<div className="h-4 bg-slate-700 rounded w-16"></div>

<div className="h-4 bg-slate-700 rounded w-16"></div>

<div className="h-4 bg-slate-700 rounded w-20"></div>

<div className="h-4 bg-slate-700 rounded w-24"></div>

</div>

))}

</div>

</div>

);

const EmptyHistoryState = () => (

<div className="text-center py-12">

<BarChart3 className="h-12 w-12 text-slate-600 mx-auto mb-4" />

<h3 className="text-lg font-semibold text-white mb-2">No Trading History</h3>

<p className="text-slate-400 mb-6">

Start trading with our signals to build your performance history.

</p>

<Button

onClick={() => navigate('/signals')}

className="bg-blue-600 hover:bg-blue-700"

>

Browse Signals

</Button>

</div>

);

**Error Boundaries & Fallbacks**

class TradeHistoryErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false, error: null };

}

static getDerivedStateFromError(error) {

return { hasError: true, error };

}

componentDidCatch(error, errorInfo) {

console.error('Trade history error:', error, errorInfo);

// Send to error reporting service

Sentry.captureException(error, { extra: errorInfo });

}

render() {

if (this.state.hasError) {

return (

<Card className="bg-slate-800/50 border-slate-700">

<CardContent className="p-8 text-center">

<AlertTriangle className="h-12 w-12 text-red-400 mx-auto mb-4" />

<h3 className="text-lg font-semibold text-white mb-2">

Unable to Load Trade History

</h3>

<p className="text-slate-400 mb-6">

There was an error loading your trading data. Please try refreshing the page.

</p>

<div className="space-x-4">

<Button

onClick={() => window.location.reload()}

className="bg-blue-600 hover:bg-blue-700"

>

Refresh Page

</Button>

<Button

onClick={() => navigate('/dashboard')}

variant="outline"

>

Back to Dashboard

</Button>

</div>

</CardContent>

</Card>

);

}

return this.props.children;

}

}

**Accessibility Implementation**

// Keyboard navigation for table

const useKeyboardNavigation = (trades, selectedIndex, setSelectedIndex) => {

useEffect(() => {

const handleKeyDown = (e) => {

switch (e.key) {

case 'ArrowDown':

e.preventDefault();

setSelectedIndex(prev => Math.min(prev + 1, trades.length - 1));

break;

case 'ArrowUp':

e.preventDefault();

setSelectedIndex(prev => Math.max(prev - 1, 0));

break;

case 'Enter':

e.preventDefault();

if (selectedIndex >= 0) {

openTradeDetail(trades[selectedIndex]);

}

break;

}

};

document.addEventListener('keydown', handleKeyDown);

return () => document.removeEventListener('keydown', handleKeyDown);

}, [trades, selectedIndex, setSelectedIndex]);

};

// Screen reader announcements

const TradeTableRow = ({ trade, isSelected, onClick }) => (

<TableRow

onClick={onClick}

className={`hover:bg-slate-700/30 ${isSelected ? 'bg-slate-700/50' : ''}`}

role="button"

tabIndex={0}

aria-label={`Trade ${trade.symbol}, ${trade.pnl >= 0 ? 'profit' : 'loss'} of $${Math.abs(trade.pnl)}`}

onKeyDown={(e) => {

if (e.key === 'Enter' || e.key === ' ') {

e.preventDefault();

onClick();

}

}}

>

{/\* Table cells \*/}

</TableRow>

);

**7. Integration Points**

**Navigation & Routing**

// Route configuration

const tradeHistoryRoutes = {

'/orders-history': {

component: OrdersHistory,

preload: ['tradeHistory', 'userStats']

},

'/orders-history/analytics': AdvancedAnalytics,

'/orders-history/export': ExportWizard,

'/orders-history/trade/:id': TradeDetailView

};

// Navigation hooks

const useTradeHistoryNavigation = () => {

const navigate = useNavigate();

const location = useLocation();

const navigateToTrade = useCallback((tradeId: string) => {

navigate(`/orders-history/trade/${tradeId}`, {

state: {

returnTo: location.pathname,

filters: getCurrentFilters()

}

});

}, [navigate, location]);

const navigateToAnalytics = useCallback(() => {

navigate('/orders-history/analytics', {

state: { source: 'history-table' }

});

}, [navigate]);

const backToDashboard = useCallback(() => {

navigate('/dashboard', {

state: {

source: 'trade-history',

highlightMetric: 'trade-performance'

}

});

}, [navigate]);

return {

navigateToTrade,

navigateToAnalytics,

backToDashboard

};

};

**Cross-Component State Sharing**

// Shared trade state management

export const useGlobalTradeState = () => {

const historyStore = useHistoricalTradesStore();

const dashboardStore = useDashboardStore();

const portfolioStore = usePortfolioStore();

// Sync trade closure with other stores

const syncTradeClose = useCallback((closedTrade: ClosedTrade) => {

historyStore.addClosedTrade(closedTrade);

dashboardStore.updateTradeMetrics();

portfolioStore.removePosition(closedTrade.symbol);

// Update global performance metrics

window.dispatchEvent(new CustomEvent('trade-closed', {

detail: { trade: closedTrade }

}));

}, [historyStore, dashboardStore, portfolioStore]);

return { syncTradeClose };

};

// Event listeners for real-time updates

useEffect(() => {

const handleTradeUpdate = (event) => {

const { trade } = event.detail;

historyStore.updateTrade(trade);

};

window.addEventListener('trade-updated', handleTradeUpdate);

return () => window.removeEventListener('trade-updated', handleTradeUpdate);

}, [historyStore]);

**8. Testing Strategy**

**Unit Tests**

// Component tests

describe('OrdersHistory Component', () => {

const mockTrades = [

{

id: '1',

symbol: 'AAPL',

pnl: 500,

pnlPercent: 5.5,

// ... other properties

}

];

beforeEach(() => {

render(

<MemoryRouter>

<OrdersHistory />

</MemoryRouter>

);

});

test('displays summary metrics correctly', () => {

expect(screen.getByText('Total Trades')).toBeInTheDocument();

expect(screen.getByText('Total P&L')).toBeInTheDocument();

expect(screen.getByText('Win Rate')).toBeInTheDocument();

});

test('filters trades by date range', async () => {

const dateFilter = screen.getByLabelText(/date range/i);

fireEvent.change(dateFilter, { target: { value: '2025-01-01,2025-06-01' } });

await waitFor(() => {

expect(screen.getByText('AAPL')).toBeInTheDocument();

});

});

test('sorts table by column', async () => {

const pnlHeader = screen.getByText('Final P&L');

fireEvent.click(pnlHeader);

await waitFor(() => {

const firstRow = screen.getAllByRole('row')[1];

expect(firstRow).toHaveTextContent('AAPL');

});

});

});

// Store tests

describe('HistoricalTradesStore', () => {

test('calculates win rate correctly', () => {

const { result } = renderHook(() => useHistoricalTradesStore());

act(() => {

result.current.setTrades([

{ pnl: 100 }, { pnl: -50 }, { pnl: 200 }

]);

});

expect(result.current.winRate).toBe(66.67);

});

test('applies filters correctly', () => {

const { result } = renderHook(() => useHistoricalTradesStore());

act(() => {

result.current.updateFilters({

status: ['Target Hit']

});

});

expect(result.current.filteredTrades).toHaveLength(2);

});

});

**Integration Tests**

// API integration tests

describe('Trade History API', () => {

beforeEach(() => {

server.use(

rest.get('/api/trades/history', (req, res, ctx) => {

return res(ctx.json({ trades: mockTrades }));

})

);

});

test('loads trade history on mount', async () => {

render(<OrdersHistory />);

await waitFor(() => {

expect(screen.getByText('AAPL')).toBeInTheDocument();

});

});

test('handles API errors gracefully', async () => {

server.use(

rest.get('/api/trades/history', (req, res, ctx) => {

return res(ctx.status(500));

})

);

render(<OrdersHistory />);

await waitFor(() => {

expect(screen.getByText(/unable to load/i)).toBeInTheDocument();

});

});

});

**Mock Data**

export const mockClosedTrades: ClosedTrade[] = [

{

id: 'trade-1',

symbol: 'AAPL',

name: 'Apple Inc.',

entryPrice: 160.02,

exitPrice: 172.45,

shares: 50,

pnl: 621.50,

pnlPercent: 7.8,

score: 88,

closedDate: '2025-06-08',

reasonForClosing: 'Target Hit',

sector: 'Technology',

openedAt: '2025-06-01T10:30:00Z',

closedAt: '2025-06-08T15:45:00Z',

holdingPeriod: 7

},

// ... more mock data

];

**9. Charts & Data Visualizations**

**Performance Chart Components**

// Monthly P&L Chart

const MonthlyPnLChart = ({ trades }) => {

const monthlyData = useMemo(() => {

return trades.reduce((acc, trade) => {

const month = format(new Date(trade.closedAt), 'yyyy-MM');

acc[month] = (acc[month] || 0) + trade.pnl;

return acc;

}, {});

}, [trades]);

const chartData = Object.entries(monthlyData).map(([month, pnl]) => ({

month: format(new Date(month), 'MMM yyyy'),

pnl: Number(pnl)

}));

return (

<ResponsiveContainer width="100%" height={300}>

<BarChart data={chartData}>

<CartesianGrid strokeDasharray="3 3" stroke="#374151" />

<XAxis

dataKey="month"

stroke="#9CA3AF"

tick={{ fill: '#9CA3AF' }}

/>

<YAxis

stroke="#9CA3AF"

tick={{ fill: '#9CA3AF' }}

tickFormatter={(value) => `$${value}`}

/>

<Tooltip

contentStyle={{

backgroundColor: '#1F2937',

border: '1px solid #374151',

borderRadius: '8px'

}}

formatter={(value) => [`$${value}`, 'P&L']}

/>

<Bar

dataKey="pnl"

fill="#3B82F6"

radius={[4, 4, 0, 0]}

/>

</BarChart>

</ResponsiveContainer>

);

};

// Win Rate Gauge

const WinRateGauge = ({ winRate }) => {

const gaugeData = [

{ name: 'Win Rate', value: winRate, fill: '#10B981' },

{ name: 'Loss Rate', value: 100 - winRate, fill: '#374151' }

];

return (

<ResponsiveContainer width="100%" height={200}>

<PieChart>

<Pie

data={gaugeData}

cx="50%"

cy="50%"

startAngle={180}

endAngle={0}

innerRadius={60}

outerRadius={80}

dataKey="value"

/>

<text

x="50%"

y="50%"

textAnchor="middle"

dominantBaseline="middle"

fontSize="24"

fontWeight="bold"

fill="#FFFFFF"

>

{winRate.toFixed(1)}%

</text>

</PieChart>

</ResponsiveContainer>

);

};

**Real-time Chart Updates**

// Animated counter for metrics

const AnimatedMetric = ({ value, duration = 1000, formatter = (v) => v }) => {

const [displayValue, setDisplayValue] = useState(0);

useEffect(() => {

let startTime;

let animationFrame;

const animate = (timestamp) => {

if (!startTime) startTime = timestamp;

const progress = Math.min((timestamp - startTime) / duration, 1);

const easedProgress = 1 - Math.pow(1 - progress, 3); // ease-out cubic

setDisplayValue(value \* easedProgress);

if (progress < 1) {

animationFrame = requestAnimationFrame(animate);

}

};

animationFrame = requestAnimationFrame(animate);

return () => cancelAnimationFrame(animationFrame);

}, [value, duration]);

return <span>{formatter(displayValue)}</span>;

};

// Usage in summary cards

const SummaryMetricCard = ({ title, value, color, formatter }) => (

<Card className="bg-slate-800/50 backdrop-blur-sm border-slate-700">

<CardContent className="p-6">

<p className="text-slate-400 text-sm mb-1">{title}</p>

<p className={`font-bold text-2xl text-${color}-400`}>

<AnimatedMetric value={value} formatter={formatter} />

</p>

</CardContent>

</Card>

);

**10. Visual Data Elements**

**Progress Indicators & Status Badges**

// Score badge with color coding

const ScoreBadge = ({ score }) => {

const getScoreColor = (score) => {

if (score >= 90) return 'bg-emerald-600 text-white';

if (score >= 80) return 'bg-blue-600 text-white';

if (score >= 70) return 'bg-amber-600 text-white';

return 'bg-red-600 text-white';

};

return (

<Badge className={`${getScoreColor(score)} font-semibold`}>

{score}/100

</Badge>

);

};

// Reason for closing badge

const ReasonBadge = ({ reason }) => {

const reasonConfig = {

'Target Hit': { color: 'bg-emerald-600', icon: Target },

'Stop Loss': { color: 'bg-red-600', icon: AlertTriangle },

'Manual Exit': { color: 'bg-blue-600', icon: Hand },

'Reversal': { color: 'bg-orange-600', icon: RotateCcw },

'Time Decay': { color: 'bg-purple-600', icon: Clock }

};

const config = reasonConfig[reason] || { color: 'bg-gray-600', icon: Circle };

const Icon = config.icon;

return (

<Badge className={`${config.color} text-white flex items-center gap-1`}>

<Icon className="h-3 w-3" />

{reason}

</Badge>

);

};

// P&L indicator with trend

const PnLIndicator = ({ pnl, pnlPercent }) => {

const isProfit = pnl >= 0;

const TrendIcon = isProfit ? TrendingUp : TrendingDown;

const colorClass = isProfit ? 'text-emerald-400' : 'text-red-400';

return (

<div className={`flex items-center gap-1 ${colorClass} font-semibold`}>

<TrendIcon className="h-4 w-4" />

<span>{isProfit ? '+' : ''}${pnl.toFixed(0)}</span>

<span className="text-sm opacity-75">

({isProfit ? '+' : ''}{pnlPercent.toFixed(1)}%)

</span>

</div>

);

};

**Visual Hierarchy & Typography**

// Consistent typography scale

const typographyClasses = {

pageTitle: 'text-3xl font-bold text-white',

sectionTitle: 'text-xl font-semibold text-white',

cardTitle: 'text-lg font-medium text-white',

metric: 'text-2xl font-bold',

label: 'text-sm text-slate-400',

body: 'text-base text-slate-300',

caption: 'text-xs text-slate-500'

};

// Color-coded status system

const statusColors = {

profit: {

bg: 'bg-emerald-500/10',

border: 'border-emerald-500/30',

text: 'text-emerald-400'

},

loss: {

bg: 'bg-red-500/10',

border: 'border-red-500/30',

text: 'text-red-400'

},

neutral: {

bg: 'bg-slate-500/10',

border: 'border-slate-500/30',

text: 'text-slate-400'

}

};

**11. Security & Validation**

**Input Validation Schemas**

import { z } from 'zod';

// Trade filter validation

const TradeFilterSchema = z.object({

dateRange: z.object({

start: z.date(),

end: z.date()

}).refine(data => data.start <= data.end, {

message: "Start date must be before end date"

}),

status: z.array(z.enum(['Target Hit', 'Stop Loss', 'Manual Exit', 'Reversal', 'Time Decay'])),

minAmount: z.number().min(0),

maxAmount: z.number().min(0),

symbols: z.array(z.string().max(10))

});

// Export request validation

const ExportRequestSchema = z.object({

format: z.enum(['csv', 'pdf']),

dateRange: z.object({

start: z.string().datetime(),

end: z.string().datetime()

}),

includeCharts: z.boolean().optional(),

trades: z.array(z.string().uuid()).max(1000) // Limit export size

});

// Trade detail validation

const TradeDetailSchema = z.object({

id: z.string().uuid(),

symbol: z.string().min(1).max(10),

entryPrice: z.number().positive(),

exitPrice: z.number().positive(),

shares: z.number().int().positive(),

pnl: z.number(),

score: z.number().min(0).max(100)

});

**Authentication & Authorization**

// Protected route wrapper

const ProtectedTradeHistory = () => {

const { user, loading } = useAuth();

const navigate = useNavigate();

useEffect(() => {

if (!loading && !user) {

navigate('/', {

state: {

message: 'Please sign in to view your trading history',

redirectTo: '/orders-history'

}

});

}

}, [user, loading, navigate]);

if (loading) return <HistoryTableSkeleton />;

if (!user) return null;

return (

<TradeHistoryErrorBoundary userId={user.id}>

<OrdersHistory />

</TradeHistoryErrorBoundary>

);

};

// Data access validation

const useSecureTradeData = () => {

const { user } = useAuth();

const fetchUserTrades = useCallback(async (filters) => {

if (!user?.id) {

throw new Error('User not authenticated');

}

const response = await fetch('/api/trades/history', {

method: 'POST',

headers: {

'Authorization': `Bearer ${await user.getIdToken()}`,

'Content-Type': 'application/json'

},

body: JSON.stringify({

...filters,

userId: user.id // Server validates this matches token

})

});

if (!response.ok) {

throw new Error(`Failed to fetch trades: ${response.status}`);

}

return response.json();

}, [user]);

return { fetchUserTrades };

};

**Data Sanitization**

// XSS prevention for dynamic content

const sanitizeTradeData = (trade: ClosedTrade): ClosedTrade => ({

...trade,

symbol: DOMPurify.sanitize(trade.symbol),

name: DOMPurify.sanitize(trade.name),

reasonForClosing: DOMPurify.sanitize(trade.reasonForClosing)

});

// Rate limiting for exports

const useExportRateLimit = () => {

const [exportCount, setExportCount] = useState(0);

const [resetTime, setResetTime] = useState(Date.now() + 3600000); // 1 hour

const canExport = exportCount < 5 && Date.now() < resetTime;

const trackExport = () => {

if (Date.now() >= resetTime) {

setExportCount(1);

setResetTime(Date.now() + 3600000);

} else {

setExportCount(prev => prev + 1);

}

};

return { canExport, trackExport, remaining: 5 - exportCount };

};

**12. Environment & Configuration**

**Environment Variables**

// .env.local

NEXT\_PUBLIC\_API\_URL=https://api.kurzora.com

NEXT\_PUBLIC\_WS\_URL=wss://ws.kurzora.com

SUPABASE\_URL=https://your-project.supabase.co

SUPABASE\_ANON\_KEY=your\_anon\_key

STRIPE\_PUBLISHABLE\_KEY=pk\_test\_...

SENTRY\_DSN=https://...@sentry.io/...

// Development

NODE\_ENV=development

NEXT\_PUBLIC\_DEBUG\_MODE=true

NEXT\_PUBLIC\_MOCK\_DATA=true

// Production

NODE\_ENV=production

NEXT\_PUBLIC\_DEBUG\_MODE=false

NEXT\_PUBLIC\_MOCK\_DATA=false

**Feature Flags**

// Feature flag configuration

const featureFlags = {

ADVANCED\_ANALYTICS: process.env.NODE\_ENV !== 'production',

REAL\_TIME\_UPDATES: true,

EXPORT\_PDF: true,

TRADE\_NOTES: false, // Coming soon

PERFORMANCE\_CHARTS: true,

BULK\_ACTIONS: false

};

// Feature flag hook

const useFeatureFlag = (flag: keyof typeof featureFlags) => {

return featureFlags[flag] ?? false;

};

// Conditional rendering

const TradeHistoryActions = () => {

const showAdvancedAnalytics = useFeatureFlag('ADVANCED\_ANALYTICS');

const showBulkActions = useFeatureFlag('BULK\_ACTIONS');

return (

<div className="flex gap-2">

{showAdvancedAnalytics && (

<Button onClick={openAnalytics}>

Advanced Analytics

</Button>

)}

{showBulkActions && (

<Button onClick={openBulkActions}>

Bulk Actions

</Button>

)}

</div>

);

};

**Performance Monitoring**

// Performance tracking

const usePagePerformance = () => {

useEffect(() => {

const observer = new PerformanceObserver((list) => {

for (const entry of list.getEntries()) {

if (entry.entryType === 'navigation') {

// Track page load time

analytics.track('Page Load Time', {

page: 'orders-history',

loadTime: entry.loadEventEnd - entry.loadEventStart,

domContentLoaded: entry.domContentLoadedEventEnd - entry.domContentLoadedEventStart

});

}

}

});

observer.observe({ entryTypes: ['navigation'] });

return () => observer.disconnect();

}, []);

};

// Error monitoring with Sentry

import \* as Sentry from '@sentry/nextjs';

const TradeHistoryWithMonitoring = () => {

const handleError = useCallback((error: Error, errorInfo: any) => {

Sentry.withScope((scope) => {

scope.setTag('component', 'OrdersHistory');

scope.setContext('errorInfo', errorInfo);

scope.setUser({ id: user?.id });

Sentry.captureException(error);

});

}, [user]);

return (

<ErrorBoundary onError={handleError}>

<OrdersHistory />

</ErrorBoundary>

);

};

**13. Cross-Screen Data Flow**

**Real-time Data Synchronization**

// WebSocket connection for real-time updates

const useTradeUpdates = () => {

const historyStore = useHistoricalTradesStore();

const { user } = useAuth();

useEffect(() => {

if (!user) return;

const ws = new WebSocket(`${process.env.NEXT\_PUBLIC\_WS\_URL}/trades/${user.id}`);

ws.onmessage = (event) => {

const data = JSON.parse(event.data);

switch (data.type) {

case 'TRADE\_CLOSED':

historyStore.addClosedTrade(data.trade);

toast.success(`Trade ${data.trade.symbol} closed with ${data.trade.pnl >= 0 ? 'profit' : 'loss'}`);

break;

case 'TRADE\_UPDATED':

historyStore.updateTrade(data.trade);

break;

case 'BULK\_UPDATE':

historyStore.refreshTrades();

break;

}

};

return () => ws.close();

}, [user, historyStore]);

};

// Shared state synchronization

const useGlobalStateSync = () => {

const historyStore = useHistoricalTradesStore();

const dashboardStore = useDashboardStore();

const portfolioStore = usePortfolioStore();

// Listen for trade closures from other components

useEffect(() => {

const handleTradeClose = (event: CustomEvent) => {

const { trade } = event.detail;

// Update all relevant stores

historyStore.addClosedTrade(trade);

dashboardStore.updateMetrics();

portfolioStore.removePosition(trade.symbol);

// Invalidate relevant queries

queryClient.invalidateQueries(['dashboard-metrics']);

queryClient.invalidateQueries(['portfolio-positions']);

};

window.addEventListener('trade-closed', handleTradeClose);

return () => window.removeEventListener('trade-closed', handleTradeClose);

}, [historyStore, dashboardStore, portfolioStore]);

// Broadcast trade history changes

const broadcastHistoryUpdate = useCallback((trades: ClosedTrade[]) => {

window.dispatchEvent(new CustomEvent('history-updated', {

detail: { trades }

}));

}, []);

return { broadcastHistoryUpdate };

};

**Cache Management & Invalidation**

// Intelligent cache invalidation

const useCacheManagement = () => {

const queryClient = useQueryClient();

const invalidateTradeData = useCallback((tradeId?: string) => {

if (tradeId) {

// Specific trade update

queryClient.invalidateQueries(['trade', tradeId]);

} else {

// Global trade data update

queryClient.invalidateQueries(['tradeHistory']);

queryClient.invalidateQueries(['dashboard-metrics']);

queryClient.invalidateQueries(['portfolio-summary']);

}

}, [queryClient]);

const prefetchRelatedData = useCallback(async (trade: ClosedTrade) => {

// Prefetch related symbol data

queryClient.prefetchQuery({

queryKey: ['symbol', trade.symbol],

queryFn: () => fetchSymbolData(trade.symbol)

});

// Prefetch sector performance

queryClient.prefetchQuery({

queryKey: ['sector-performance', trade.sector],

queryFn: () => fetchSectorPerformance(trade.sector)

});

}, [queryClient]);

return { invalidateTradeData, prefetchRelatedData };

};

// Optimistic updates for better UX

const useOptimisticTradeUpdates = () => {

const queryClient = useQueryClient();

const optimisticTradeClose = useCallback(async (trade: ClosedTrade) => {

// Optimistically update cache

queryClient.setQueryData(['tradeHistory'], (oldData: any) => ({

...oldData,

trades: [trade, ...oldData.trades]

}));

try {

// Actual API call

await fetch('/api/trades/close', {

method: 'POST',

body: JSON.stringify(trade)

});

} catch (error) {

// Revert on error

queryClient.invalidateQueries(['tradeHistory']);

throw error;

}

}, [queryClient]);

return { optimisticTradeClose };

};

**Implementation Priority:**

1. Core table functionality and filtering
2. Summary metrics calculation
3. Export functionality
4. Real-time updates
5. Advanced analytics
6. Performance optimizations