**🎯 Kurzora Settings Page - Complete UI Analysis**

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

**1. UI Components & Layout**

**Interactive Elements**

**Primary Settings Components:**

* **NotificationSettingsCard** (email, telegram, push toggles + digest frequency)
* **LanguageLocaleCard** (language badges, timezone dropdown)
* **SecuritySettingsCard** (password change, 2FA toggle, login activity)
* **APISettingsCard** (API key management, auto-trading toggle, broker status)
* **UIPreferencesCard** (display preferences, theme settings)
* **DataPrivacyCard** (future: data export, account deletion)

**Form Controls & Interactive Elements:**

* Toggle switches for boolean preferences
* Dropdown selectors for enums
* Language selection badges with flags
* API key input with regeneration
* Security action buttons
* Real-time status indicators

**Enhanced React + TypeScript Component Structure**

// Complete Settings Page Architecture

<Layout>

<div className={`max-w-4xl mx-auto px-4 sm:px-6 lg:px-8 py-8 ${language === 'ar' ? 'rtl' : 'ltr'}`}>

{/\* Header Section \*/}

<div className="mb-8">

<h1 className="text-3xl font-bold text-white mb-2">{t('settings.title')}</h1>

<p className="text-slate-400">{t('settings.subtitle')}</p>

{/\* Settings Save Banner \*/}

<SettingsSaveBanner

hasUnsavedChanges={hasUnsavedChanges}

onSave={handleSaveAll}

onDiscard={handleDiscardChanges}

/>

</div>

{/\* Quick Actions Bar \*/}

<QuickActionsBar

onExportData={handleExportData}

onImportSettings={handleImportSettings}

onResetToDefaults={handleResetToDefaults}

/>

{/\* Settings Sections \*/}

<div className="space-y-6">

{/\* Notification Settings \*/}

<NotificationSettingsCard

settings={notificationSettings}

onChange={updateNotificationSettings}

onTestNotification={handleTestNotification}

isLoading={isUpdatingNotifications}

/>

{/\* Language & Locale \*/}

<LanguageLocaleCard

currentLanguage={language}

currentTimezone={timezone}

supportedLanguages={supportedLanguages}

supportedTimezones={supportedTimezones}

onLanguageChange={handleLanguageChange}

onTimezoneChange={handleTimezoneChange}

/>

{/\* Security Settings \*/}

<SecuritySettingsCard

user={user}

securitySettings={securitySettings}

loginSessions={loginSessions}

onChangePassword={handleChangePassword}

on2FAToggle={handle2FAToggle}

onRevokeSession={handleRevokeSession}

onViewAllSessions={handleViewAllSessions}

/>

{/\* API & Auto-Trading \*/}

<APISettingsCard

apiKeys={apiKeys}

autoTradingSettings={autoTradingSettings}

brokerConnections={brokerConnections}

onGenerateAPIKey={handleGenerateAPIKey}

onRevokeAPIKey={handleRevokeAPIKey}

onAutoTradingToggle={handleAutoTradingToggle}

onBrokerConnect={handleBrokerConnect}

/>

{/\* UI Preferences \*/}

<UIPreferencesCard

uiSettings={uiSettings}

themeSettings={themeSettings}

onUISettingChange={handleUISettingChange}

onThemeChange={handleThemeChange}

/>

</div>

{/\* Modals \*/}

<ChangePasswordModal

open={isChangePasswordOpen}

onOpenChange={setIsChangePasswordOpen}

onSuccess={handlePasswordChangeSuccess}

/>

<Setup2FAModal

open={is2FASetupOpen}

onOpenChange={setIs2FASetupOpen}

onSuccess={handle2FASetupSuccess}

/>

<ExportDataModal

open={isExportModalOpen}

onOpenChange={setIsExportModalOpen}

onExport={handleDataExport}

/>

</div>

</Layout>

**Enhanced NotificationSettingsCard Component**

interface NotificationSettingsProps {

settings: NotificationSettings;

onChange: (settings: NotificationSettings) => void;

onTestNotification: (type: 'email' | 'telegram' | 'push') => Promise<void>;

isLoading?: boolean;

}

const NotificationSettingsCard: React.FC<NotificationSettingsProps> = ({

settings,

onChange,

onTestNotification,

isLoading = false

}) => {

const { t } = useLanguage();

const [testingNotification, setTestingNotification] = useState<string | null>(null);

const handleTestNotification = async (type: 'email' | 'telegram' | 'push') => {

setTestingNotification(type);

try {

await onTestNotification(type);

toast.success(`${type} notification sent successfully!`);

} catch (error) {

toast.error(`Failed to send ${type} notification`);

} finally {

setTestingNotification(null);

}

};

return (

<Card className="bg-slate-800/50 backdrop-blur-sm border-slate-700 hover:bg-slate-800/70 transition-all duration-300">

<CardHeader>

<CardTitle className="text-white flex items-center justify-between">

<div className="flex items-center">

<Bell className="h-5 w-5 mr-2 text-blue-400" />

{t('settings.notifications')}

</div>

{isLoading && <Spinner className="h-4 w-4" />}

</CardTitle>

</CardHeader>

<CardContent className="space-y-6">

{/\* Email Alerts \*/}

<SettingRow

id="email-alerts"

label={t('settings.emailAlerts')}

description="Get notified about high-score signals via email"

checked={settings.emailAlerts}

onChange={(checked) => onChange({ ...settings, emailAlerts: checked })}

icon={Mail}

actions={

<Button

variant="ghost"

size="sm"

onClick={() => handleTestNotification('email')}

disabled={!settings.emailAlerts || testingNotification === 'email'}

className="text-blue-400 hover:text-blue-300"

>

{testingNotification === 'email' ? <Spinner className="h-3 w-3" /> : 'Test'}

</Button>

}

/>

{/\* Telegram Alerts \*/}

<SettingRow

id="telegram-alerts"

label={t('settings.telegramAlerts')}

description="Instant signals delivery via Telegram bot"

checked={settings.telegramAlerts}

onChange={(checked) => onChange({ ...settings, telegramAlerts: checked })}

icon={MessageSquare}

actions={

<Button

variant="ghost"

size="sm"

onClick={() => handleTestNotification('telegram')}

disabled={!settings.telegramAlerts || testingNotification === 'telegram'}

className="text-blue-400 hover:text-blue-300"

>

{testingNotification === 'telegram' ? <Spinner className="h-3 w-3" /> : 'Test'}

</Button>

}

/>

{/\* Push Notifications \*/}

<SettingRow

id="push-notifications"

label={t('settings.pushNotifications')}

description="Browser push notifications for urgent signals"

checked={settings.pushNotifications}

onChange={(checked) => onChange({ ...settings, pushNotifications: checked })}

icon={Smartphone}

actions={

<Button

variant="ghost"

size="sm"

onClick={() => handleTestNotification('push')}

disabled={!settings.pushNotifications || testingNotification === 'push'}

className="text-blue-400 hover:text-blue-300"

>

{testingNotification === 'push' ? <Spinner className="h-3 w-3" /> : 'Test'}

</Button>

}

/>

{/\* Digest Frequency \*/}

<div className="pt-4 border-t border-slate-600">

<Label className="text-slate-300 text-sm font-medium">{t('settings.digestFreq')}</Label>

<Select

value={settings.digestFrequency}

onValueChange={(value) => onChange({ ...settings, digestFrequency: value as any })}

>

<SelectTrigger className="mt-2 bg-slate-700 border-slate-600 text-white">

<SelectValue />

</SelectTrigger>

<SelectContent>

<SelectItem value="realtime">

<div className="flex items-center space-x-2">

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

<span>Real-Time</span>

</div>

</SelectItem>

<SelectItem value="hourly">Hourly Digest</SelectItem>

<SelectItem value="daily">Daily Digest</SelectItem>

<SelectItem value="weekly">Weekly Summary</SelectItem>

</SelectContent>

</Select>

</div>

{/\* Signal Score Threshold for Notifications \*/}

<div>

<Label className="text-slate-300 text-sm font-medium">

Minimum Signal Score for Notifications

</Label>

<div className="mt-2">

<Slider

value={[settings.minScoreThreshold]}

onValueChange={([value]) => onChange({ ...settings, minScoreThreshold: value })}

max={100}

min={50}

step={5}

className="w-full"

/>

<div className="flex justify-between text-xs text-slate-400 mt-1">

<span>50</span>

<span className="text-white font-medium">{settings.minScoreThreshold}</span>

<span>100</span>

</div>

</div>

</div>

</CardContent>

</Card>

);

};

**Responsive Design Considerations**

/\* Mobile First Approach \*/

.settings-container {

@apply max-w-4xl mx-auto px-4 sm:px-6 lg:px-8 py-8;

}

.settings-grid {

@apply grid grid-cols-1 lg:grid-cols-2 gap-6;

}

.settings-card {

@apply bg-slate-800/50 backdrop-blur-sm border-slate-700 hover:bg-slate-800/70 transition-all duration-300;

}

.setting-row {

@apply flex flex-col sm:flex-row sm:items-center sm:justify-between space-y-2 sm:space-y-0;

}

.setting-row-mobile {

@apply flex flex-col space-y-3;

}

@media (max-width: 640px) {

.settings-actions {

@apply flex flex-col space-y-2 w-full;

}

.settings-button {

@apply w-full justify-center;

}

}

**Loading States & Error Handling**

// Loading skeleton for settings cards

const SettingsCardSkeleton: React.FC = () => (

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

<CardHeader>

<div className="flex items-center space-x-2">

<Skeleton className="h-5 w-5" />

<Skeleton className="h-6 w-32" />

</div>

</CardHeader>

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

{[...Array(3)].map((\_, i) => (

<div key={i} className="flex items-center justify-between">

<div className="space-y-1">

<Skeleton className="h-4 w-24" />

<Skeleton className="h-3 w-40" />

</div>

<Skeleton className="h-6 w-11" />

</div>

))}

</CardContent>

</Card>

);

// Error boundary for settings sections

const SettingsErrorBoundary: React.FC<{ children: React.ReactNode }> = ({ children }) => (

<ErrorBoundary

fallback={

<Card className="bg-red-900/20 border-red-800/50">

<CardContent className="p-6 text-center">

<AlertTriangle className="h-8 w-8 text-red-400 mx-auto mb-4" />

<p className="text-red-400 mb-4">Failed to load settings section</p>

<Button onClick={() => window.location.reload()} variant="outline">

Retry

</Button>

</CardContent>

</Card>

}

>

{children}

</ErrorBoundary>

);

**2. State Management (Zustand)**

**Settings Store Structure**

interface SettingsState {

// Settings Data

notificationSettings: NotificationSettings;

securitySettings: SecuritySettings;

uiSettings: UISettings;

apiSettings: APISettings;

// UI State

isLoading: boolean;

hasUnsavedChanges: boolean;

activeSection: string | null;

modals: {

changePassword: boolean;

setup2FA: boolean;

exportData: boolean;

importSettings: boolean;

};

// Actions

updateNotificationSettings: (settings: Partial<NotificationSettings>) => void;

updateSecuritySettings: (settings: Partial<SecuritySettings>) => void;

updateUISettings: (settings: Partial<UISettings>) => void;

updateAPISettings: (settings: Partial<APISettings>) => void;

// Modal Actions

openModal: (modal: keyof SettingsState['modals']) => void;

closeModal: (modal: keyof SettingsState['modals']) => void;

// Persistence Actions

saveAllSettings: () => Promise<void>;

loadSettings: () => Promise<void>;

discardChanges: () => void;

resetToDefaults: () => Promise<void>;

// Utility Actions

markUnsaved: () => void;

markSaved: () => void;

setActiveSection: (section: string | null) => void;

}

const useSettingsStore = create<SettingsState>((set, get) => ({

// Initial State

notificationSettings: {

emailAlerts: true,

telegramAlerts: true,

pushNotifications: false,

digestFrequency: 'daily',

minScoreThreshold: 80,

quietHours: { enabled: false, start: '22:00', end: '07:00' }

},

securitySettings: {

twoFactorEnabled: false,

sessionTimeout: 60,

deviceTrustEnabled: true,

loginNotifications: true,

apiKeyRotationDays: 90

},

uiSettings: {

theme: 'dark',

language: 'en',

timezone: 'UTC',

showPnL: true,

compactMode: false,

showAnimations: true,

autoRefresh: true,

refreshInterval: 30

},

apiSettings: {

autoTrading: false,

apiKeys: [],

brokerConnections: [],

webhookUrl: null,

rateLimit: 100

},

isLoading: false,

hasUnsavedChanges: false,

activeSection: null,

modals: {

changePassword: false,

setup2FA: false,

exportData: false,

importSettings: false

},

// Update Actions with Optimistic Updates

updateNotificationSettings: (settings) => {

set((state) => ({

notificationSettings: { ...state.notificationSettings, ...settings },

hasUnsavedChanges: true

}));

// Debounced auto-save

debouncedAutoSave();

},

updateSecuritySettings: (settings) => {

set((state) => ({

securitySettings: { ...state.securitySettings, ...settings },

hasUnsavedChanges: true

}));

debouncedAutoSave();

},

updateUISettings: (settings) => {

set((state) => ({

uiSettings: { ...state.uiSettings, ...settings },

hasUnsavedChanges: true

}));

debouncedAutoSave();

},

updateAPISettings: (settings) => {

set((state) => ({

apiSettings: { ...state.apiSettings, ...settings },

hasUnsavedChanges: true

}));

debouncedAutoSave();

},

// Modal Management

openModal: (modal) => set((state) => ({

modals: { ...state.modals, [modal]: true }

})),

closeModal: (modal) => set((state) => ({

modals: { ...state.modals, [modal]: false }

})),

// Persistence Actions

saveAllSettings: async () => {

const state = get();

set({ isLoading: true });

try {

await settingsAPI.updateUserSettings({

notifications: state.notificationSettings,

security: state.securitySettings,

ui: state.uiSettings,

api: state.apiSettings

});

set({ hasUnsavedChanges: false, isLoading: false });

toast.success('Settings saved successfully');

} catch (error) {

set({ isLoading: false });

toast.error('Failed to save settings');

throw error;

}

},

loadSettings: async () => {

set({ isLoading: true });

try {

const settings = await settingsAPI.getUserSettings();

set({

notificationSettings: settings.notifications,

securitySettings: settings.security,

uiSettings: settings.ui,

apiSettings: settings.api,

hasUnsavedChanges: false,

isLoading: false

});

} catch (error) {

set({ isLoading: false });

toast.error('Failed to load settings');

}

},

discardChanges: () => {

// Reload from last saved state

get().loadSettings();

set({ hasUnsavedChanges: false });

},

resetToDefaults: async () => {

set({ isLoading: true });

try {

await settingsAPI.resetToDefaults();

await get().loadSettings();

toast.success('Settings reset to defaults');

} catch (error) {

set({ isLoading: false });

toast.error('Failed to reset settings');

}

},

markUnsaved: () => set({ hasUnsavedChanges: true }),

markSaved: () => set({ hasUnsavedChanges: false }),

setActiveSection: (section) => set({ activeSection: section })

}));

// Debounced auto-save to prevent excessive API calls

const debouncedAutoSave = debounce(async () => {

const { hasUnsavedChanges, saveAllSettings } = useSettingsStore.getState();

if (hasUnsavedChanges) {

try {

await saveAllSettings();

} catch (error) {

console.error('Auto-save failed:', error);

}

}

}, 2000);

**Local State Patterns**

// Component-level state for immediate UI feedback

const NotificationSettingsCard: React.FC = () => {

const { notificationSettings, updateNotificationSettings } = useSettingsStore();

// Local state for immediate UI feedback

const [localSettings, setLocalSettings] = useState(notificationSettings);

const [testingStatus, setTestingStatus] = useState<Record<string, boolean>>({});

// Sync with store when it changes

useEffect(() => {

setLocalSettings(notificationSettings);

}, [notificationSettings]);

// Optimistic update pattern

const handleSettingChange = useCallback((key: string, value: any) => {

// Update local state immediately for UI responsiveness

setLocalSettings(prev => ({ ...prev, [key]: value }));

// Update store (which triggers API call)

updateNotificationSettings({ [key]: value });

}, [updateNotificationSettings]);

return (

// Component JSX using localSettings for immediate feedback

// and store state for persistence

);

};

**3. API Contracts & Integration**

**Settings API Endpoints**

// API Contract Definitions

interface SettingsAPI {

// GET /api/user/settings

getUserSettings(): Promise<UserSettingsResponse>;

// PUT /api/user/settings

updateUserSettings(settings: UpdateUserSettingsRequest): Promise<void>;

// POST /api/user/settings/reset

resetToDefaults(): Promise<void>;

// POST /api/user/settings/export

exportUserData(): Promise<ExportDataResponse>;

// POST /api/user/settings/import

importUserSettings(data: ImportDataRequest): Promise<void>;

// GET /api/user/settings/activity

getSettingsActivity(): Promise<SettingsActivityResponse>;

// API Key Management

// GET /api/user/api-keys

getAPIKeys(): Promise<APIKeyResponse[]>;

// POST /api/user/api-keys

generateAPIKey(request: GenerateAPIKeyRequest): Promise<APIKeyResponse>;

// DELETE /api/user/api-keys/:id

revokeAPIKey(keyId: string): Promise<void>;

// Security endpoints

// POST /api/user/password/change

changePassword(request: ChangePasswordRequest): Promise<void>;

// POST /api/user/2fa/setup

setup2FA(): Promise<Setup2FAResponse>;

// POST /api/user/2fa/verify

verify2FA(request: Verify2FARequest): Promise<void>;

// POST /api/user/2fa/disable

disable2FA(request: Disable2FARequest): Promise<void>;

// Notification testing

// POST /api/user/notifications/test

testNotification(request: TestNotificationRequest): Promise<void>;

}

// TypeScript Interface Definitions

interface UserSettingsResponse {

notifications: NotificationSettings;

security: SecuritySettings;

ui: UISettings;

api: APISettings;

lastModified: string;

version: number;

}

interface NotificationSettings {

emailAlerts: boolean;

telegramAlerts: boolean;

pushNotifications: boolean;

digestFrequency: 'realtime' | 'hourly' | 'daily' | 'weekly';

minScoreThreshold: number;

quietHours: {

enabled: boolean;

start: string; // HH:mm format

end: string; // HH:mm format

timezone: string;

};

channels: {

email: string;

telegram: string | null;

webhook: string | null;

};

}

interface SecuritySettings {

twoFactorEnabled: boolean;

sessionTimeout: number; // minutes

deviceTrustEnabled: boolean;

loginNotifications: boolean;

apiKeyRotationDays: number;

trustedDevices: TrustedDevice[];

loginSessions: LoginSession[];

}

interface UISettings {

theme: 'light' | 'dark' | 'auto';

language: 'en' | 'ar' | 'de';

timezone: string;

showPnL: boolean;

compactMode: boolean;

showAnimations: boolean;

autoRefresh: boolean;

refreshInterval: number; // seconds

numberFormat: 'us' | 'eu' | 'in';

dateFormat: 'iso' | 'us' | 'eu';

}

interface APISettings {

autoTrading: boolean;

apiKeys: APIKey[];

brokerConnections: BrokerConnection[];

webhookUrl: string | null;

rateLimit: number; // requests per minute

autoTradingRules: {

minScore: number;

maxRiskPerTrade: number;

maxDailyRisk: number;

allowedMarkets: string[];

};

}

interface APIKey {

id: string;

name: string;

key: string; // Masked in responses

permissions: string[];

lastUsed: string | null;

createdAt: string;

expiresAt: string | null;

isActive: boolean;

}

// Request/Response Schemas

interface UpdateUserSettingsRequest {

notifications?: Partial<NotificationSettings>;

security?: Partial<SecuritySettings>;

ui?: Partial<UISettings>;

api?: Partial<APISettings>;

}

interface ChangePasswordRequest {

currentPassword: string;

newPassword: string;

confirmPassword: string;

}

interface Setup2FAResponse {

qrCode: string; // Base64 encoded QR code

secret: string; // Backup secret

backupCodes: string[];

}

interface TestNotificationRequest {

type: 'email' | 'telegram' | 'push' | 'webhook';

message?: string;

}

// Error Response Format

interface SettingsAPIError {

code: string;

message: string;

field?: string;

details?: Record<string, any>;

}

// API Implementation

export const settingsAPI: SettingsAPI = {

async getUserSettings(): Promise<UserSettingsResponse> {

const response = await fetch('/api/user/settings', {

headers: {

'Authorization': `Bearer ${getToken()}`,

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

},

});

if (!response.ok) {

throw new SettingsAPIError(await response.json());

}

return response.json();

},

async updateUserSettings(settings: UpdateUserSettingsRequest): Promise<void> {

const response = await fetch('/api/user/settings', {

method: 'PUT',

headers: {

'Authorization': `Bearer ${getToken()}`,

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

},

body: JSON.stringify(settings),

});

if (!response.ok) {

throw new SettingsAPIError(await response.json());

}

},

async testNotification(request: TestNotificationRequest): Promise<void> {

const response = await fetch('/api/user/notifications/test', {

method: 'POST',

headers: {

'Authorization': `Bearer ${getToken()}`,

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

},

body: JSON.stringify(request),

});

if (!response.ok) {

throw new SettingsAPIError(await response.json());

}

},

async generateAPIKey(request: GenerateAPIKeyRequest): Promise<APIKeyResponse> {

const response = await fetch('/api/user/api-keys', {

method: 'POST',

headers: {

'Authorization': `Bearer ${getToken()}`,

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

},

body: JSON.stringify(request),

});

if (!response.ok) {

throw new SettingsAPIError(await response.json());

}

return response.json();

},

async changePassword(request: ChangePasswordRequest): Promise<void> {

const response = await fetch('/api/user/password/change', {

method: 'POST',

headers: {

'Authorization': `Bearer ${getToken()}`,

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

},

body: JSON.stringify(request),

});

if (!response.ok) {

throw new SettingsAPIError(await response.json());

}

}

};

**Pagination & Filtering Parameters**

// Settings Activity Log with Pagination

interface SettingsActivityRequest {

page?: number;

limit?: number;

startDate?: string;

endDate?: string;

actionType?: 'update' | 'reset' | 'export' | 'import';

section?: 'notifications' | 'security' | 'ui' | 'api';

}

interface SettingsActivityResponse {

activities: SettingsActivity[];

pagination: {

page: number;

limit: number;

total: number;

totalPages: number;

};

}

interface SettingsActivity {

id: string;

action: string;

section: string;

changes: Record<string, { from: any; to: any }>;

timestamp: string;

ipAddress: string;

userAgent: string;

}

**4. Performance & Optimization**

**Lazy Loading Strategies**

// Lazy load heavy modal components

const ChangePasswordModal = lazy(() => import('./modals/ChangePasswordModal'));

const Setup2FAModal = lazy(() => import('./modals/Setup2FAModal'));

const ExportDataModal = lazy(() => import('./modals/ExportDataModal'));

const ImportSettingsModal = lazy(() => import('./modals/ImportSettingsModal'));

// Lazy load settings sections

const NotificationSettingsCard = lazy(() => import('./cards/NotificationSettingsCard'));

const SecuritySettingsCard = lazy(() => import('./cards/SecuritySettingsCard'));

const APISettingsCard = lazy(() => import('./cards/APISettingsCard'));

// Conditional loading based on user permissions

const AdminSettingsCard = lazy(() => import('./cards/AdminSettingsCard'));

// Settings page with lazy components

const Settings: React.FC = () => {

const { user, isAdmin } = useAuth();

return (

<Layout>

<Suspense fallback={<SettingsPageSkeleton />}>

<div className="settings-container">

<Suspense fallback={<SettingsCardSkeleton />}>

<NotificationSettingsCard />

</Suspense>

<Suspense fallback={<SettingsCardSkeleton />}>

<SecuritySettingsCard />

</Suspense>

<Suspense fallback={<SettingsCardSkeleton />}>

<APISettingsCard />

</Suspense>

{isAdmin() && (

<Suspense fallback={<SettingsCardSkeleton />}>

<AdminSettingsCard />

</Suspense>

)}

</div>

</Suspense>

</Layout>

);

};

**Memoization Opportunities**

// Memoized settings components to prevent unnecessary re-renders

const NotificationSettingsCard = React.memo<NotificationSettingsProps>(({

settings,

onChange,

onTestNotification,

isLoading

}) => {

// Memoize expensive calculations

const notificationStatus = useMemo(() => {

return {

totalEnabled: Object.values(settings).filter(Boolean).length,

hasQuietHours: settings.quietHours?.enabled,

nextDigest: calculateNextDigestTime(settings.digestFrequency)

};

}, [settings]);

// Memoize callback functions

const handleToggle = useCallback((key: keyof NotificationSettings) => {

return (checked: boolean) => {

onChange({ ...settings, [key]: checked });

};

}, [settings, onChange]);

const memoizedTestNotification = useCallback((type: string) => {

return () => onTestNotification(type as any);

}, [onTestNotification]);

return (

// Component JSX

);

});

// Memoized setting row component

const SettingRow = React.memo<{

id: string;

label: string;

description?: string;

checked: boolean;

onChange: (checked: boolean) => void;

disabled?: boolean;

icon?: React.ComponentType<{ className?: string }>;

}>(({ id, label, description, checked, onChange, disabled, icon: Icon }) => {

return (

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

<div className="flex items-center space-x-3">

{Icon && <Icon className="h-4 w-4 text-slate-400" />}

<div>

<Label

htmlFor={id}

className="text-slate-300 cursor-pointer"

>

{label}

</Label>

{description && (

<p className="text-xs text-slate-400">{description}</p>

)}

</div>

</div>

<Switch

id={id}

checked={checked}

onCheckedChange={onChange}

disabled={disabled}

aria-describedby={description ? `${id}-description` : undefined}

/>

{description && (

<span id={`${id}-description`} className="sr-only">

{description}

</span>

)}

</div>

);

});

**Bundle Splitting**

// Route-based splitting

const SettingsPage = lazy(() => import('./pages/Settings'));

const AdminSettingsPage = lazy(() => import('./pages/AdminSettings'));

// Feature-based splitting

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

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

// Third-party library splitting

const qrCodeGenerator = () => import('qrcode-generator');

const cryptoUtils = () => import('./utils/crypto');

// Critical CSS inlining for settings page

const criticalStyles = `

.settings-container { /\* Critical styles \*/ }

.settings-card { /\* Critical card styles \*/ }

`;

**Caching Strategies**

// React Query for settings caching

const useSettings = () => {

return useQuery({

queryKey: ['user', 'settings'],

queryFn: settingsAPI.getUserSettings,

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

cacheTime: 10 \* 60 \* 1000, // 10 minutes

refetchOnWindowFocus: false,

refetchOnMount: false,

});

};

// Mutation with optimistic updates

const useUpdateSettings = () => {

const queryClient = useQueryClient();

return useMutation({

mutationFn: settingsAPI.updateUserSettings,

onMutate: async (newSettings) => {

// Cancel outgoing refetches

await queryClient.cancelQueries(['user', 'settings']);

// Snapshot previous value

const previousSettings = queryClient.getQueryData(['user', 'settings']);

// Optimistically update

queryClient.setQueryData(['user', 'settings'], (old: any) => ({

...old,

...newSettings

}));

return { previousSettings };

},

onError: (err, newSettings, context) => {

// Rollback on error

queryClient.setQueryData(['user', 'settings'], context?.previousSettings);

},

onSettled: () => {

// Refetch after mutation

queryClient.invalidateQueries(['user', 'settings']);

},

});

};

// Service worker caching for offline support

if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/sw.js').then(() => {

// Cache settings API responses

self.addEventListener('fetch', (event) => {

if (event.request.url.includes('/api/user/settings')) {

event.respondWith(

caches.open('settings-cache').then((cache) => {

return cache.match(event.request).then((response) => {

return response || fetch(event.request).then((fetchResponse) => {

cache.put(event.request, fetchResponse.clone());

return fetchResponse;

});

});

})

);

}

});

});

}

**5. Database Schema**

**PostgreSQL Tables**

-- Users table (enhanced for settings)

CREATE TABLE users (

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

email VARCHAR(255) UNIQUE NOT NULL,

password\_hash VARCHAR(255),

name VARCHAR(255) NOT NULL,

-- Settings columns

notification\_settings JSONB DEFAULT '{}',

security\_settings JSONB DEFAULT '{}',

ui\_settings JSONB DEFAULT '{}',

api\_settings JSONB DEFAULT '{}',

-- Metadata

settings\_version INTEGER DEFAULT 1,

last\_settings\_update TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

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

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- User API Keys table

CREATE TABLE user\_api\_keys (

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

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

name VARCHAR(255) NOT NULL,

key\_hash VARCHAR(255) NOT NULL, -- Hashed API key

key\_prefix VARCHAR(10) NOT NULL, -- First 8 chars for display

permissions TEXT[] DEFAULT '{}',

last\_used TIMESTAMP WITH TIME ZONE,

expires\_at TIMESTAMP WITH TIME ZONE,

is\_active BOOLEAN DEFAULT true,

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

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Two-Factor Authentication table

CREATE TABLE user\_2fa (

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

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

secret VARCHAR(255) NOT NULL,

backup\_codes JSONB DEFAULT '[]',

is\_enabled BOOLEAN DEFAULT false,

last\_used TIMESTAMP WITH TIME ZONE,

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

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- User Sessions table

CREATE TABLE user\_sessions (

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

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

session\_token VARCHAR(255) UNIQUE NOT NULL,

ip\_address INET,

user\_agent TEXT,

device\_fingerprint VARCHAR(255),

is\_trusted BOOLEAN DEFAULT false,

expires\_at TIMESTAMP WITH TIME ZONE NOT NULL,

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

last\_activity TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Settings Activity Log table

CREATE TABLE settings\_activity (

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

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

action VARCHAR(50) NOT NULL, -- 'update', 'reset', 'export', 'import'

section VARCHAR(50) NOT NULL, -- 'notifications', 'security', 'ui', 'api'

old\_values JSONB,

new\_values JSONB,

ip\_address INET,

user\_agent TEXT,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Notification Queue table

CREATE TABLE notification\_queue (

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

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

type VARCHAR(50) NOT NULL, -- 'email', 'telegram', 'push', 'webhook'

recipient VARCHAR(255) NOT NULL,

subject VARCHAR(255),

content TEXT NOT NULL,

metadata JSONB DEFAULT '{}',

status VARCHAR(20) DEFAULT 'pending', -- 'pending', 'sent', 'failed', 'cancelled'

attempts INTEGER DEFAULT 0,

max\_attempts INTEGER DEFAULT 3,

next\_attempt TIMESTAMP WITH TIME ZONE,

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

sent\_at TIMESTAMP WITH TIME ZONE,

failed\_at TIMESTAMP WITH TIME ZONE,

error\_message TEXT

);

-- Broker Connections table

CREATE TABLE broker\_connections (

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

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

broker\_name VARCHAR(50) NOT NULL, -- 'ibkr', 'td\_ameritrade', etc.

connection\_status VARCHAR(20) DEFAULT 'disconnected',

credentials\_encrypted BYTEA, -- Encrypted broker credentials

last\_connected TIMESTAMP WITH TIME ZONE,

auto\_trading\_enabled BOOLEAN DEFAULT false,

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

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Indexes for performance

CREATE INDEX idx\_users\_email ON users(email);

CREATE INDEX idx\_user\_api\_keys\_user\_id ON user\_api\_keys(user\_id);

CREATE INDEX idx\_user\_api\_keys\_active ON user\_api\_keys(user\_id, is\_active);

CREATE INDEX idx\_user\_sessions\_user\_id ON user\_sessions(user\_id);

CREATE INDEX idx\_user\_sessions\_token ON user\_sessions(session\_token);

CREATE INDEX idx\_user\_sessions\_expires ON user\_sessions(expires\_at);

CREATE INDEX idx\_settings\_activity\_user\_id ON settings\_activity(user\_id);

CREATE INDEX idx\_settings\_activity\_created ON settings\_activity(created\_at);

CREATE INDEX idx\_notification\_queue\_status ON notification\_queue(status, next\_attempt);

CREATE INDEX idx\_notification\_queue\_user\_id ON notification\_queue(user\_id);

CREATE INDEX idx\_broker\_connections\_user\_id ON broker\_connections(user\_id);

-- Constraints

ALTER TABLE user\_api\_keys ADD CONSTRAINT unique\_api\_key\_name\_per\_user

UNIQUE(user\_id, name);

ALTER TABLE user\_2fa ADD CONSTRAINT one\_2fa\_per\_user

UNIQUE(user\_id);

-- Row Level Security

ALTER TABLE users ENABLE ROW LEVEL SECURITY;

ALTER TABLE user\_api\_keys ENABLE ROW LEVEL SECURITY;

ALTER TABLE user\_2fa ENABLE ROW LEVEL SECURITY;

ALTER TABLE user\_sessions ENABLE ROW LEVEL SECURITY;

ALTER TABLE settings\_activity ENABLE ROW LEVEL SECURITY;

ALTER TABLE notification\_queue ENABLE ROW LEVEL SECURITY;

ALTER TABLE broker\_connections ENABLE ROW LEVEL SECURITY;

-- RLS Policies

CREATE POLICY users\_own\_data ON users

FOR ALL USING (auth.uid() = id);

CREATE POLICY user\_api\_keys\_own\_data ON user\_api\_keys

FOR ALL USING (auth.uid() = user\_id);

CREATE POLICY user\_2fa\_own\_data ON user\_2fa

FOR ALL USING (auth.uid() = user\_id);

CREATE POLICY user\_sessions\_own\_data ON user\_sessions

FOR ALL USING (auth.uid() = user\_id);

CREATE POLICY settings\_activity\_own\_data ON settings\_activity

FOR ALL USING (auth.uid() = user\_id);

CREATE POLICY notification\_queue\_own\_data ON notification\_queue

FOR ALL USING (auth.uid() = user\_id);

CREATE POLICY broker\_connections\_own\_data ON broker\_connections

FOR ALL USING (auth.uid() = user\_id);

**Migration Scripts**

-- Migration 001: Initial settings schema

CREATE OR REPLACE FUNCTION update\_settings\_timestamp()

RETURNS TRIGGER AS $$

BEGIN

NEW.updated\_at = NOW();

NEW.last\_settings\_update = NOW();

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER update\_users\_timestamp

BEFORE UPDATE ON users

FOR EACH ROW

EXECUTE FUNCTION update\_settings\_timestamp();

-- Migration 002: Add settings validation

CREATE OR REPLACE FUNCTION validate\_notification\_settings(settings JSONB)

RETURNS BOOLEAN AS $$

BEGIN

-- Validate notification settings structure

IF NOT (settings ? 'emailAlerts' AND

settings ? 'telegramAlerts' AND

settings ? 'pushNotifications') THEN

RETURN FALSE;

END IF;

-- Validate digest frequency

IF NOT (settings->>'digestFrequency' IN ('realtime', 'hourly', 'daily', 'weekly')) THEN

RETURN FALSE;

END IF;

-- Validate score threshold

IF NOT ((settings->>'minScoreThreshold')::INTEGER BETWEEN 50 AND 100) THEN

RETURN FALSE;

END IF;

RETURN TRUE;

END;

$$ LANGUAGE plpgsql;

-- Add validation constraint

ALTER TABLE users ADD CONSTRAINT valid\_notification\_settings

CHECK (validate\_notification\_settings(notification\_settings));

**6. User Experience**

**Loading States & Skeleton Screens**

// Settings page skeleton

const SettingsPageSkeleton: React.FC = () => (

<Layout>

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

<div className="mb-8">

<Skeleton className="h-8 w-48 mb-2" />

<Skeleton className="h-4 w-96" />

</div>

<div className="space-y-6">

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

<SettingsCardSkeleton key={i} />

))}

</div>

</div>

</Layout>

);

// Individual settings card skeleton

const SettingsCardSkeleton: React.FC = () => (

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

<CardHeader>

<div className="flex items-center space-x-2">

<Skeleton className="h-5 w-5" />

<Skeleton className="h-6 w-32" />

</div>

</CardHeader>

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

{[...Array(3)].map((\_, i) => (

<div key={i} className="flex items-center justify-between">

<div className="space-y-1">

<Skeleton className="h-4 w-24" />

<Skeleton className="h-3 w-40" />

</div>

<Skeleton className="h-6 w-11" />

</div>

))}

</CardContent>

</Card>

);

// Progressive loading for settings sections

const ProgressiveSettingsLoader: React.FC = () => {

const [loadedSections, setLoadedSections] = useState<string[]>([]);

useEffect(() => {

const sections = ['notifications', 'security', 'api', 'ui'];

sections.forEach((section, index) => {

setTimeout(() => {

setLoadedSections(prev => [...prev, section]);

}, index \* 300); // Stagger loading by 300ms

});

}, []);

return (

<div className="space-y-6">

{loadedSections.includes('notifications') ? (

<NotificationSettingsCard />

) : (

<SettingsCardSkeleton />

)}

{loadedSections.includes('security') ? (

<SecuritySettingsCard />

) : (

<SettingsCardSkeleton />

)}

{/\* Continue for other sections \*/}

</div>

);

};

**Error Boundaries & Fallback UI**

// Settings-specific error boundary

interface SettingsErrorBoundaryState {

hasError: boolean;

error: Error | null;

errorInfo: ErrorInfo | null;

}

class SettingsErrorBoundary extends Component<

{ children: ReactNode; section?: string },

SettingsErrorBoundaryState

> {

constructor(props: any) {

super(props);

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

}

static getDerivedStateFromError(error: Error): SettingsErrorBoundaryState {

return { hasError: true, error, errorInfo: null };

}

componentDidCatch(error: Error, errorInfo: ErrorInfo) {

this.setState({ errorInfo });

// Log error to monitoring service

if (process.env.NODE\_ENV === 'production') {

console.error('Settings Error:', error, errorInfo);

// Send to error tracking service

}

}

render() {

if (this.state.hasError) {

return (

<Card className="bg-red-900/20 border-red-800/50">

<CardContent className="p-6 text-center">

<AlertTriangle className="h-8 w-8 text-red-400 mx-auto mb-4" />

<h3 className="text-red-400 font-medium mb-2">

{this.props.section ?

`Failed to load ${this.props.section} settings` :

'Settings section unavailable'

}

</h3>

<p className="text-slate-400 text-sm mb-4">

There was an error loading this settings section. You can try refreshing or contact support.

</p>

<div className="space-x-2">

<Button

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

variant="outline"

size="sm"

>

Refresh Page

</Button>

<Button

onClick={() => this.setState({ hasError: false })}

variant="ghost"

size="sm"

>

Try Again

</Button>

</div>

</CardContent>

</Card>

);

}

return this.props.children;

}

}

// Usage wrapper for each settings section

const SafeSettingsSection: React.FC<{

section: string;

children: ReactNode

}> = ({ section, children }) => (

<SettingsErrorBoundary section={section}>

<Suspense fallback={<SettingsCardSkeleton />}>

{children}

</Suspense>

</SettingsErrorBoundary>

);

**Accessibility Considerations**

// Accessible setting toggle component

const AccessibleSettingRow: React.FC<{

id: string;

label: string;

description?: string;

checked: boolean;

onChange: (checked: boolean) => void;

disabled?: boolean;

icon?: React.ComponentType<{ className?: string }>;

}> = ({ id, label, description, checked, onChange, disabled, icon: Icon }) => {

return (

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

<div className="flex items-center space-x-3">

{Icon && <Icon className="h-4 w-4 text-slate-400" />}

<div>

<Label

htmlFor={id}

className="text-slate-300 cursor-pointer"

>

{label}

</Label>

{description && (

<p className="text-xs text-slate-400">{description}</p>

)}

</div>

</div>

<Switch

id={id}

checked={checked}

onCheckedChange={onChange}

disabled={disabled}

aria-describedby={description ? `${id}-description` : undefined}

/>

{description && (

<span id={`${id}-description`} className="sr-only">

{description}

</span>

)}

</div>

);

};

// Keyboard navigation for settings sections

const useSettingsKeyboardNavigation = () => {

useEffect(() => {

const handleKeyDown = (event: KeyboardEvent) => {

// Tab navigation enhancement

if (event.key === 'Tab') {

// Ensure logical tab order through settings sections

}

// Save shortcut

if (event.key === 's' && (event.ctrlKey || event.metaKey)) {

event.preventDefault();

useSettingsStore.getState().saveAllSettings();

}

// Escape to discard changes

if (event.key === 'Escape') {

const { hasUnsavedChanges, discardChanges } = useSettingsStore.getState();

if (hasUnsavedChanges) {

if (confirm('Discard unsaved changes?')) {

discardChanges();

}

}

}

};

document.addEventListener('keydown', handleKeyDown);

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

}, []);

};

// Screen reader announcements for settings changes

const useSettingsAnnouncements = () => {

const announce = (message: string) => {

const announcement = document.createElement('div');

announcement.setAttribute('aria-live', 'polite');

announcement.setAttribute('aria-atomic', 'true');

announcement.className = 'sr-only';

announcement.textContent = message;

document.body.appendChild(announcement);

setTimeout(() => document.body.removeChild(announcement), 1000);

};

const announceSettingChange = (setting: string, newValue: any) => {

announce(`${setting} updated to ${newValue}`);

};

const announceSettingsSaved = () => {

announce('All settings saved successfully');

};

return { announceSettingChange, announceSettingsSaved };

};

**Animation & Transition Requirements**

// Smooth transitions for settings changes

const AnimatedSettingsCard: React.FC<{ children: ReactNode }> = ({ children }) => (

<motion.div

initial={{ opacity: 0, y: 20 }}

animate={{ opacity: 1, y: 0 }}

exit={{ opacity: 0, y: -20 }}

transition={{ duration: 0.3, ease: 'easeInOut' }}

>

{children}

</motion.div>

);

// Success animation for saving settings

const SaveSuccessAnimation: React.FC<{ visible: boolean }> = ({ visible }) => (

<AnimatePresence>

{visible && (

<motion.div

initial={{ scale: 0, opacity: 0 }}

animate={{ scale: 1, opacity: 1 }}

exit={{ scale: 0, opacity: 0 }}

className="fixed top-4 right-4 bg-green-600 text-white px-4 py-2 rounded-md shadow-lg z-50"

>

<div className="flex items-center space-x-2">

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

<span>Settings saved!</span>

</div>

</motion.div>

)}

</AnimatePresence>

);

// Micro-interactions for switches

const AnimatedSwitch: React.FC<SwitchProps> = ({ checked, onCheckedChange, ...props }) => (

<motion.div

whileTap={{ scale: 0.95 }}

transition={{ type: "spring", stiffness: 400, damping: 17 }}

>

<Switch

checked={checked}

onCheckedChange={onCheckedChange}

{...props}

/>

</motion.div>

);

**7. Integration Points**

**Cross-Page Settings Propagation**

// Settings event system for cross-page updates

export const settingsEventEmitter = new EventTarget();

export const SettingsEvents = {

LANGUAGE\_CHANGED: 'settings:language\_changed',

THEME\_CHANGED: 'settings:theme\_changed',

NOTIFICATIONS\_UPDATED: 'settings:notifications\_updated',

TIMEZONE\_CHANGED: 'settings:timezone\_changed',

API\_KEY\_GENERATED: 'settings:api\_key\_generated',

SECURITY\_UPDATED: 'settings:security\_updated'

} as const;

// Settings change propagation

export const emitSettingsEvent = (eventType: string, data: any) => {

settingsEventEmitter.dispatchEvent(

new CustomEvent(eventType, { detail: data })

);

};

// Global settings listener hook

export const useGlobalSettingsSync = () => {

const languageStore = useLanguageStore();

const themeStore = useThemeStore();

const notificationStore = useNotificationStore();

useEffect(() => {

const handleLanguageChange = (event: CustomEvent) => {

const { language, timezone } = event.detail;

languageStore.setLanguage(language);

languageStore.setTimezone(timezone);

};

const handleThemeChange = (event: CustomEvent) => {

const { theme } = event.detail;

themeStore.setTheme(theme);

document.documentElement.setAttribute('data-theme', theme);

};

const handleNotificationsUpdate = (event: CustomEvent) => {

const notificationSettings = event.detail;

notificationStore.updateSettings(notificationSettings);

};

settingsEventEmitter.addEventListener(SettingsEvents.LANGUAGE\_CHANGED, handleLanguageChange);

settingsEventEmitter.addEventListener(SettingsEvents.THEME\_CHANGED, handleThemeChange);

settingsEventEmitter.addEventListener(SettingsEvents.NOTIFICATIONS\_UPDATED, handleNotificationsUpdate);

return () => {

settingsEventEmitter.removeEventListener(SettingsEvents.LANGUAGE\_CHANGED, handleLanguageChange);

settingsEventEmitter.removeEventListener(SettingsEvents.THEME\_CHANGED, handleThemeChange);

settingsEventEmitter.removeEventListener(SettingsEvents.NOTIFICATIONS\_UPDATED, handleNotificationsUpdate);

};

}, []);

};

**Navigation & Routing Integration**

// Settings navigation configuration

const settingsRoutes = {

settings: '/settings',

notifications: '/settings/notifications',

security: '/settings/security',

api: '/settings/api',

privacy: '/settings/privacy',

billing: '/settings/billing' // Future

};

// Settings section navigation

const useSettingsNavigation = () => {

const navigate = useNavigate();

const location = useLocation();

const navigateToSection = (section: keyof typeof settingsRoutes) => {

const route = settingsRoutes[section];

navigate(route);

};

const getCurrentSection = () => {

const path = location.pathname;

return Object.entries(settingsRoutes).find(([\_, route]) =>

path === route

)?.[0] || 'settings';

};

return { navigateToSection, getCurrentSection };

};

// Settings breadcrumb navigation

const SettingsBreadcrumb: React.FC = () => {

const { getCurrentSection } = useSettingsNavigation();

const { t } = useLanguage();

const currentSection = getCurrentSection();

const breadcrumbItems = [

{ label: t('nav.dashboard'), href: '/dashboard' },

{ label: t('nav.settings'), href: '/settings' },

];

if (currentSection !== 'settings') {

breadcrumbItems.push({

label: t(`settings.${currentSection}`),

href: settingsRoutes[currentSection as keyof typeof settingsRoutes]

});

}

return (

<nav className="flex mb-6" aria-label="Breadcrumb">

<ol className="flex items-center space-x-2">

{breadcrumbItems.map((item, index) => (

<li key={item.href} className="flex items-center">

{index > 0 && (

<ChevronRight className="h-4 w-4 text-slate-400 mx-2" />

)}

<Link

to={item.href}

className={`text-sm ${

index === breadcrumbItems.length - 1

? 'text-white font-medium'

: 'text-slate-400 hover:text-white'

}`}

>

{item.label}

</Link>

</li>

))}

</ol>

</nav>

);

};

**Shared Components & State**

// Shared settings hook for cross-component access

export const useSharedSettings = () => {

const {

notificationSettings,

uiSettings,

securitySettings,

updateNotificationSettings,

updateUISettings,

updateSecuritySettings

} = useSettingsStore();

const isFeatureEnabled = (feature: string): boolean => {

switch (feature) {

case 'darkMode':

return uiSettings.theme === 'dark';

case 'autoRefresh':

return uiSettings.autoRefresh;

case '2fa':

return securitySettings.twoFactorEnabled;

default:

return false;

}

};

const getNotificationPreference = (type: 'email' | 'telegram' | 'push'): boolean => {

return notificationSettings[`${type}Alerts` as keyof typeof notificationSettings] as boolean;

};

return {

settings: { notificationSettings, uiSettings, securitySettings },

actions: { updateNotificationSettings, updateUISettings, updateSecuritySettings },

helpers: { isFeatureEnabled, getNotificationPreference }

};

};

// Settings-aware components used across the app

export const SettingsAwareComponent: React.FC = () => {

const { settings, helpers } = useSharedSettings();

// Auto-refresh based on settings

useEffect(() => {

if (helpers.isFeatureEnabled('autoRefresh')) {

const interval = setInterval(() => {

// Refresh data

}, settings.uiSettings.refreshInterval \* 1000);

return () => clearInterval(interval);

}

}, [settings.uiSettings.autoRefresh, settings.uiSettings.refreshInterval]);

return (

// Component JSX that adapts to settings

);

};

**8. Testing Strategy**

**Unit Test Requirements**

// Settings store tests

describe('useSettingsStore', () => {

beforeEach(() => {

useSettingsStore.getState().reset();

});

it('should update notification settings', () => {

const { updateNotificationSettings, notificationSettings } = useSettingsStore.getState();

updateNotificationSettings({ emailAlerts: false });

expect(useSettingsStore.getState().notificationSettings.emailAlerts).toBe(false);

expect(useSettingsStore.getState().hasUnsavedChanges).toBe(true);

});

it('should save all settings', async () => {

const mockAPI = vi.spyOn(settingsAPI, 'updateUserSettings').mockResolvedValue();

const { saveAllSettings, updateNotificationSettings } = useSettingsStore.getState();

updateNotificationSettings({ emailAlerts: false });

await saveAllSettings();

expect(mockAPI).toHaveBeenCalledWith({

notifications: expect.objectContaining({ emailAlerts: false })

});

expect(useSettingsStore.getState().hasUnsavedChanges).toBe(false);

});

it('should handle save errors', async () => {

const mockAPI = vi.spyOn(settingsAPI, 'updateUserSettings').mockRejectedValue(new Error('Save failed'));

const { saveAllSettings } = useSettingsStore.getState();

await expect(saveAllSettings()).rejects.toThrow('Save failed');

expect(useSettingsStore.getState().hasUnsavedChanges).toBe(true);

});

});

// Component tests

describe('NotificationSettingsCard', () => {

const mockProps = {

settings: {

emailAlerts: true,

telegramAlerts: false,

pushNotifications: false,

digestFrequency: 'daily',

minScoreThreshold: 80

},

onChange: vi.fn(),

onTestNotification: vi.fn().mockResolvedValue(undefined)

};

it('should render all notification options', () => {

render(<NotificationSettingsCard {...mockProps} />);

expect(screen.getByLabelText(/email alerts/i)).toBeInTheDocument();

expect(screen.getByLabelText(/telegram alerts/i)).toBeInTheDocument();

expect(screen.getByLabelText(/push notifications/i)).toBeInTheDocument();

});

it('should toggle email alerts', async () => {

render(<NotificationSettingsCard {...mockProps} />);

const emailToggle = screen.getByLabelText(/email alerts/i);

await userEvent.click(emailToggle);

expect(mockProps.onChange).toHaveBeenCalledWith({

...mockProps.settings,

emailAlerts: false

});

});

it('should test notifications', async () => {

render(<NotificationSettingsCard {...mockProps} />);

const testButton = screen.getByRole('button', { name: /test email/i });

await userEvent.click(testButton);

expect(mockProps.onTestNotification).toHaveBeenCalledWith('email');

});

it('should disable test button when setting is off', () => {

const propsWithEmailOff = {

...mockProps,

settings: { ...mockProps.settings, emailAlerts: false }

};

render(<NotificationSettingsCard {...propsWithEmailOff} />);

const testButton = screen.getByRole('button', { name: /test email/i });

expect(testButton).toBeDisabled();

});

});

// API integration tests

describe('settingsAPI', () => {

beforeEach(() => {

fetchMock.resetMocks();

});

it('should fetch user settings', async () => {

const mockSettings = {

notifications: { emailAlerts: true },

security: { twoFactorEnabled: false },

ui: { theme: 'dark' },

api: { autoTrading: false }

};

fetchMock.mockResponseOnce(JSON.stringify(mockSettings));

const result = await settingsAPI.getUserSettings();

expect(result).toEqual(mockSettings);

expect(fetchMock).toHaveBeenCalledWith('/api/user/settings', {

headers: expect.objectContaining({

'Authorization': expect.stringContaining('Bearer'),

'Content-Type': 'application/json'

})

});

});

it('should handle API errors', async () => {

fetchMock.mockResponseOnce(

JSON.stringify({ code: 'INVALID\_REQUEST', message: 'Invalid settings' }),

{ status: 400 }

);

await expect(settingsAPI.updateUserSettings({})).rejects.toThrow();

});

});

**Integration Test Scenarios**

// Full settings flow integration test

describe('Settings Integration', () => {

it('should complete full settings update flow', async () => {

// Mock API responses

fetchMock

.mockResponseOnce(JSON.stringify(mockInitialSettings)) // GET settings

.mockResponseOnce('', { status: 200 }); // PUT settings

// Render settings page

render(<Settings />, { wrapper: TestWrapper });

// Wait for settings to load

await waitFor(() => {

expect(screen.getByText('Email Alerts')).toBeInTheDocument();

});

// Toggle email alerts

const emailToggle = screen.getByLabelText(/email alerts/i);

await userEvent.click(emailToggle);

// Save settings

const saveButton = screen.getByRole('button', { name: /save settings/i });

await userEvent.click(saveButton);

// Verify API call

expect(fetchMock).toHaveBeenCalledWith('/api/user/settings', {

method: 'PUT',

headers: expect.any(Object),

body: JSON.stringify({

notifications: expect.objectContaining({ emailAlerts: false })

})

});

// Verify success message

await waitFor(() => {

expect(screen.getByText(/settings saved/i)).toBeInTheDocument();

});

});

it('should handle validation errors', async () => {

fetchMock.mockResponseOnce(

JSON.stringify({

code: 'VALIDATION\_ERROR',

message: 'Invalid score threshold',

field: 'minScoreThreshold'

}),

{ status: 400 }

);

render(<Settings />, { wrapper: TestWrapper });

// Try to save invalid settings

const saveButton = screen.getByRole('button', { name: /save settings/i });

await userEvent.click(saveButton);

// Verify error message

await waitFor(() => {

expect(screen.getByText(/invalid score threshold/i)).toBeInTheDocument();

});

});

});

**Mock Data Structures**

// Mock settings data for testing

export const mockSettingsData = {

notifications: {

emailAlerts: true,

telegramAlerts: false,

pushNotifications: true,

digestFrequency: 'daily' as const,

minScoreThreshold: 80,

quietHours: {

enabled: false,

start: '22:00',

end: '07:00',

timezone: 'UTC'

},

channels: {

email: 'user@example.com',

telegram: null,

webhook: null

}

},

security: {

twoFactorEnabled: false,

sessionTimeout: 60,

deviceTrustEnabled: true,

loginNotifications: true,

apiKeyRotationDays: 90,

trustedDevices: [],

loginSessions: []

},

ui: {

theme: 'dark' as const,

language: 'en' as const,

timezone: 'UTC',

showPnL: true,

compactMode: false,

showAnimations: true,

autoRefresh: true,

refreshInterval: 30,

numberFormat: 'us' as const,

dateFormat: 'iso' as const

},

api: {

autoTrading: false,

apiKeys: [],

brokerConnections: [],

webhookUrl: null,

rateLimit: 100,

autoTradingRules: {

minScore: 85,

maxRiskPerTrade: 2,

maxDailyRisk: 10,

allowedMarkets: ['NASDAQ', 'NYSE']

}

}

};

// Mock API key data

export const mockAPIKeyData = {

id: 'key\_123',

name: 'Trading Bot API',

key: 'sk-\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*',

permissions: ['read', 'write'],

lastUsed: '2024-01-15T10:30:00Z',

createdAt: '2024-01-01T00:00:00Z',

expiresAt: null,

isActive: true

};

// Test utilities

export const TestWrapper: React.FC<{ children: ReactNode }> = ({ children }) => (

<QueryClient>

<LanguageProvider>

<AuthProvider>

<BrowserRouter>

{children}

</BrowserRouter>

</AuthProvider>

</LanguageProvider>

</QueryClient>

);

**Edge Cases to Handle**

// Edge case testing scenarios

describe('Settings Edge Cases', () => {

it('should handle network errors gracefully', async () => {

fetchMock.mockRejectOnce(new Error('Network error'));

render(<Settings />, { wrapper: TestWrapper });

await waitFor(() => {

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

});

});

it('should handle corrupted settings data', async () => {

fetchMock.mockResponseOnce('invalid json');

render(<Settings />, { wrapper: TestWrapper });

await waitFor(() => {

expect(screen.getByText(/error loading settings/i)).toBeInTheDocument();

});

});

it('should handle concurrent settings updates', async () => {

const store = useSettingsStore.getState();

// Simulate multiple rapid updates

store.updateNotificationSettings({ emailAlerts: false });

store.updateNotificationSettings({ telegramAlerts: true });

store.updateUISettings({ theme: 'light' });

// Should still maintain correct state

expect(store.notificationSettings.emailAlerts).toBe(false);

expect(store.notificationSettings.telegramAlerts).toBe(true);

expect(store.uiSettings.theme).toBe('light');

});

it('should handle API timeout', async () => {

fetchMock.mockAbortOnce();

const store = useSettingsStore.getState();

await expect(store.saveAllSettings()).rejects.toThrow();

expect(store.hasUnsavedChanges).toBe(true);

});

});

**9. Charts & Data Visualizations**

*Note: The Settings page focuses on configuration rather than data visualization. However, there are some visual elements that enhance the user experience:*

**Progress Indicators & Visual Feedback**

// Settings completion progress

const SettingsCompletionIndicator: React.FC = () => {

const { notificationSettings, securitySettings, uiSettings, apiSettings } = useSettingsStore();

const calculateCompletionScore = useMemo(() => {

let completed = 0;

let total = 0;

// Notification settings completion

total += 4;

if (notificationSettings.emailAlerts || notificationSettings.telegramAlerts) completed += 1;

if (notificationSettings.digestFrequency !== 'realtime') completed += 1;

if (notificationSettings.minScoreThreshold >= 70) completed += 1;

if (notificationSettings.quietHours?.enabled) completed += 1;

// Security settings completion

total += 3;

if (securitySettings.twoFactorEnabled) completed += 1;

if (securitySettings.sessionTimeout <= 60) completed += 1;

if (securitySettings.deviceTrustEnabled) completed += 1;

// UI settings completion

total += 2;

if (uiSettings.language !== 'en') completed += 1; // International users

if (uiSettings.timezone !== 'UTC') completed += 1;

// API settings completion

total += 2;

if (apiSettings.apiKeys.length > 0) completed += 1;

if (apiSettings.autoTrading) completed += 1;

return Math.round((completed / total) \* 100);

}, [notificationSettings, securitySettings, uiSettings, apiSettings]);

return (

<Card className="bg-gradient-to-r from-blue-900/20 to-purple-900/20 border-blue-800/50">

<CardContent className="p-4">

<div className="flex items-center justify-between mb-2">

<span className="text-sm font-medium text-slate-300">Settings Completion</span>

<span className="text-sm text-blue-400">{calculateCompletionScore}%</span>

</div>

<div className="w-full bg-slate-700 rounded-full h-2">

<motion.div

className="bg-gradient-to-r from-blue-500 to-purple-500 h-2 rounded-full"

initial={{ width: 0 }}

animate={{ width: `${calculateCompletionScore}%` }}

transition={{ duration: 0.8, ease: "easeOut" }}

/>

</div>

<p className="text-xs text-slate-400 mt-2">

{calculateCompletionScore >= 80

? "Great! Your settings are well configured."

: "Complete more settings to enhance your experience."

}

</p>

</CardContent>

</Card>

);

};

// Security score indicator

const SecurityScoreIndicator: React.FC = () => {

const { securitySettings } = useSettingsStore();

const securityScore = useMemo(() => {

let score = 0;

if (securitySettings.twoFactorEnabled) score += 40;

if (securitySettings.sessionTimeout <= 30) score += 20;

if (securitySettings.deviceTrustEnabled) score += 15;

if (securitySettings.loginNotifications) score += 15;

if (securitySettings.apiKeyRotationDays <= 90) score += 10;

return Math.min(score, 100);

}, [securitySettings]);

const getSecurityLevel = (score: number) => {

if (score >= 80) return { level: 'High', color: 'text-green-400', bgColor: 'bg-green-400' };

if (score >= 60) return { level: 'Medium', color: 'text-yellow-400', bgColor: 'bg-yellow-400' };

return { level: 'Low', color: 'text-red-400', bgColor: 'bg-red-400' };

};

const security = getSecurityLevel(securityScore);

return (

<div className="flex items-center space-x-3">

<div className="relative w-12 h-12">

<svg className="w-12 h-12 transform -rotate-90" viewBox="0 0 36 36">

<path

className="text-slate-700"

stroke="currentColor"

strokeWidth="3"

fill="none"

d="M18 2.0845 a 15.9155 15.9155 0 0 1 0 31.831 a 15.9155 15.9155 0 0 1 0 -31.831"

/>

<motion.path

className={security.color}

stroke="currentColor"

strokeWidth="3"

fill="none"

strokeLinecap="round"

d="M18 2.0845 a 15.9155 15.9155 0 0 1 0 31.831 a 15.9155 15.9155 0 0 1 0 -31.831"

initial={{ strokeDasharray: "0 100" }}

animate={{ strokeDasharray: `${securityScore} 100` }}

transition={{ duration: 1, ease: "easeOut" }}

/>

</svg>

<div className="absolute inset-0 flex items-center justify-center">

<span className={`text-xs font-bold ${security.color}`}>

{securityScore}

</span>

</div>

</div>

<div>

<div className="flex items-center space-x-2">

<Shield className={`h-4 w-4 ${security.color}`} />

<span className={`text-sm font-medium ${security.color}`}>

{security.level} Security

</span>

</div>

<p className="text-xs text-slate-400">

{securityScore >= 80

? "Your account is well protected"

: "Consider enabling more security features"

}

</p>

</div>

</div>

);

};

**Dynamic Counters & Animated Numbers**

// Animated settings stats

const SettingsStatsDisplay: React.FC = () => {

const {

notificationSettings,

securitySettings,

apiSettings

} = useSettingsStore();

const stats = useMemo(() => [

{

label: "Active Alerts",

value: [

notificationSettings.emailAlerts,

notificationSettings.telegramAlerts,

notificationSettings.pushNotifications

].filter(Boolean).length,

max: 3,

icon: Bell,

color: "text-blue-400"

},

{

label: "API Keys",

value: apiSettings.apiKeys.filter(key => key.isActive).length,

max: 5,

icon: Key,

color: "text-yellow-400"

},

{

label: "Security Features",

value: [

securitySettings.twoFactorEnabled,

securitySettings.deviceTrustEnabled,

securitySettings.loginNotifications

].filter(Boolean).length,

max: 3,

icon: Shield,

color: "text-green-400"

}

], [notificationSettings, securitySettings, apiSettings]);

return (

<div className="grid grid-cols-1 md:grid-cols-3 gap-4 mb-6">

{stats.map((stat, index) => (

<Card key={stat.label} className="bg-slate-800/30 border-slate-700">

<CardContent className="p-4">

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

<div>

<motion.div

className={`text-2xl font-bold ${stat.color}`}

initial={{ opacity: 0, scale: 0.5 }}

animate={{ opacity: 1, scale: 1 }}

transition={{

duration: 0.5,

delay: index \* 0.1,

type: "spring",

stiffness: 100

}}

>

<CountUp

end={stat.value}

duration={0.8}

delay={index \* 0.2}

/>

<span className="text-slate-400 text-sm ml-1">/ {stat.max}</span>

</motion.div>

<p className="text-sm text-slate-400 mt-1">{stat.label}</p>

</div>

<stat.icon className={`h-8 w-8 ${stat.color}`} />

</div>

</CardContent>

</Card>

))}

</div>

);

};

// Real-time settings usage metrics

const SettingsUsageChart: React.FC = () => {

const [usageData, setUsageData] = useState<any[]>([]);

useEffect(() => {

// Fetch settings usage analytics

const fetchUsageData = async () => {

try {

const data = await settingsAPI.getSettingsUsage();

setUsageData(data.dailyUsage);

} catch (error) {

console.error('Failed to fetch usage data:', error);

}

};

fetchUsageData();

}, []);

if (usageData.length === 0) return null;

return (

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

<CardHeader>

<CardTitle className="text-white text-lg">Settings Activity (Last 7 Days)</CardTitle>

</CardHeader>

<CardContent>

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

<AreaChart data={usageData}>

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

<XAxis

dataKey="date"

stroke="#9CA3AF"

fontSize={12}

tickFormatter={(value) => format(new Date(value), 'MMM dd')}

/>

<YAxis stroke="#9CA3AF" fontSize={12} />

<Tooltip

contentStyle={{

backgroundColor: '#1F2937',

border: '1px solid #374151',

borderRadius: '8px',

color: '#F9FAFB'

}}

/>

<Area

type="monotone"

dataKey="changes"

stroke="#3B82F6"

fill="url(#settingsGradient)"

strokeWidth={2}

/>

<defs>

<linearGradient id="settingsGradient" x1="0" y1="0" x2="0" y2="1">

<stop offset="5%" stopColor="#3B82F6" stopOpacity={0.3} />

<stop offset="95%" stopColor="#3B82F6" stopOpacity={0} />

</linearGradient>

</defs>

</AreaChart>

</ResponsiveContainer>

</CardContent>

</Card>

);

};

**10. Visual Data Elements**

**Color-Coded Status Indicators**

// Settings status indicators

const SettingsStatusIndicator: React.FC<{

status: 'enabled' | 'disabled' | 'warning' | 'error';

label: string;

description?: string;

}> = ({ status, label, description }) => {

const statusConfig = {

enabled: {

color: 'text-green-400',

bgColor: 'bg-green-400/10',

borderColor: 'border-green-400/20',

icon: CheckCircle

},

disabled: {

color: 'text-slate-400',

bgColor: 'bg-slate-400/10',

borderColor: 'border-slate-400/20',

icon: XCircle

},

warning: {

color: 'text-yellow-400',

bgColor: 'bg-yellow-400/10',

borderColor: 'border-yellow-400/20',

icon: AlertTriangle

},

error: {

color: 'text-red-400',

bgColor: 'bg-red-400/10',

borderColor: 'border-red-400/20',

icon: AlertCircle

}

};

const config = statusConfig[status];

const Icon = config.icon;

return (

<div className={`flex items-center space-x-2 px-3 py-2 rounded-lg border ${config.bgColor} ${config.borderColor}`}>

<Icon className={`h-4 w-4 ${config.color}`} />

<div>

<span className={`text-sm font-medium ${config.color}`}>{label}</span>

{description && (

<p className="text-xs text-slate-400">{description}</p>

)}

</div>

</div>

);

};

// Broker connection status

const BrokerConnectionStatus: React.FC<{

connection: BrokerConnection;

}> = ({ connection }) => {

const getStatusConfig = (status: string) => {

switch (status) {

case 'connected':

return { status: 'enabled' as const, label: 'Connected', pulse: true };

case 'connecting':

return { status: 'warning' as const, label: 'Connecting...', pulse: true };

case 'disconnected':

return { status: 'disabled' as const, label: 'Disconnected', pulse: false };

case 'error':

return { status: 'error' as const, label: 'Connection Error', pulse: false };

default:

return { status: 'disabled' as const, label: 'Unknown', pulse: false };

}

};

const config = getStatusConfig(connection.status);

return (

<div className="flex items-center justify-between p-4 bg-slate-800/30 rounded-lg">

<div className="flex items-center space-x-3">

<div className="relative">

<img

src={`/brokers/${connection.broker}.svg`}

alt={connection.broker}

className="h-8 w-8"

/>

{config.pulse && (

<motion.div

className="absolute -top-1 -right-1 h-3 w-3 bg-green-400 rounded-full"

animate={{ scale: [1, 1.2, 1] }}

transition={{ repeat: Infinity, duration: 2 }}

/>

)}

</div>

<div>

<h4 className="text-white font-medium">{connection.name}</h4>

<p className="text-xs text-slate-400">{connection.broker.toUpperCase()}</p>

</div>

</div>

<SettingsStatusIndicator

status={config.status}

label={config.label}

description={connection.lastConnected ?

`Last connected: ${formatDistanceToNow(new Date(connection.lastConnected))} ago` :

undefined

}

/>

</div>

);

};

**Typography Scale & Visual Hierarchy**

// Settings typography system

const SettingsTypography = {

// Page title

pageTitle: "text-3xl font-bold text-white mb-2",

pageSubtitle: "text-slate-400 mb-8",

// Section headers

sectionTitle: "text-lg font-semibold text-white flex items-center",

sectionDescription: "text-sm text-slate-400 mt-1",

// Setting labels

settingLabel: "text-slate-300 font-medium cursor-pointer",

settingDescription: "text-xs text-slate-400 mt-1",

settingValue: "text-white font-mono text-sm",

// Status text

statusSuccess: "text-green-400 text-sm font-medium",

statusWarning: "text-yellow-400 text-sm font-medium",

statusError: "text-red-400 text-sm font-medium",

statusInfo: "text-blue-400 text-sm font-medium",

// Helper text

helperText: "text-slate-400 text-xs",

linkText: "text-blue-400 hover:text-blue-300 underline cursor-pointer",

// Form elements

formLabel: "text-slate-300 text-sm font-medium",

formValue: "text-white",

formPlaceholder: "text-slate-500",

formError: "text-red-400 text-xs mt-1"

};

// Consistent text components

const SettingsText: React.FC<{

variant: keyof typeof SettingsTypography;

children: ReactNode;

className?: string;

}> = ({ variant, children, className = "" }) => {

const baseClasses = SettingsTypography[variant];

return (

<span className={cn(baseClasses, className)}>

{children}

</span>

);

};

// Settings card with proper hierarchy

const SettingsCardWithHierarchy: React.FC<{

title: string;

description?: string;

icon: LucideIcon;

iconColor: string;

children: ReactNode;

actions?: ReactNode;

}> = ({ title, description, icon: Icon, iconColor, children, actions }) => (

<Card className="bg-slate-800/50 backdrop-blur-sm border-slate-700 hover:bg-slate-800/70 transition-all duration-300">

<CardHeader className="pb-4">

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

<div className="flex items-center space-x-3">

<Icon className={`h-5 w-5 ${iconColor}`} />

<div>

<SettingsText variant="sectionTitle">{title}</SettingsText>

{description && (

<SettingsText variant="sectionDescription">{description}</SettingsText>

)}

</div>

</div>

{actions && <div className="flex items-center space-x-2">{actions}</div>}

</div>

</CardHeader>

<CardContent className="pt-0">

{children}

</CardContent>

</Card>

);

**Visual Feedback for State Changes**

// Animated state change feedback

const StateChangeIndicator: React.FC<{

isVisible: boolean;

type: 'success' | 'error' | 'warning' | 'info';

message: string;

duration?: number;

}> = ({ isVisible, type, message, duration = 3000 }) => {

const [show, setShow] = useState(isVisible);

useEffect(() => {

if (isVisible) {

setShow(true);

const timer = setTimeout(() => setShow(false), duration);

return () => clearTimeout(timer);

}

}, [isVisible, duration]);

const typeConfig = {

success: { icon: CheckCircle, color: 'text-green-400', bgColor: 'bg-green-400/10' },

error: { icon: XCircle, color: 'text-red-400', bgColor: 'bg-red-400/10' },

warning: { icon: AlertTriangle, color: 'text-yellow-400', bgColor: 'bg-yellow-400/10' },

info: { icon: Info, color: 'text-blue-400', bgColor: 'bg-blue-400/10' }

};

const config = typeConfig[type];

const Icon = config.icon;

return (

<AnimatePresence>

{show && (

<motion.div

initial={{ opacity: 0, y: -20, scale: 0.95 }}

animate={{ opacity: 1, y: 0, scale: 1 }}

exit={{ opacity: 0, y: -20, scale: 0.95 }}

className={`fixed top-4 right-4 z-50 flex items-center space-x-2 px-4 py-3 rounded-lg border ${config.bgColor} border-slate-600 backdrop-blur-sm`}

>

<Icon className={`h-4 w-4 ${config.color}`} />

<span className="text-white text-sm font-medium">{message}</span>

</motion.div>

)}

</AnimatePresence>

);

};

// Settings value change animation

const AnimatedSettingValue: React.FC<{

value: string | number;

prefix?: string;

suffix?: string;

className?: string;

}> = ({ value, prefix = "", suffix = "", className = "" }) => {

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

const [isChanging, setIsChanging] = useState(false);

useEffect(() => {

if (value !== displayValue) {

setIsChanging(true);

const timer = setTimeout(() => {

setDisplayValue(value);

setIsChanging(false);

}, 150);

return () => clearTimeout(timer);

}

}, [value, displayValue]);

return (

<motion.span

className={cn("inline-block", className)}

animate={{

scale: isChanging ? [1, 1.05, 1] : 1,

color: isChanging ? ["#F9FAFB", "#3B82F6", "#F9FAFB"] : "#F9FAFB"

}}

transition={{ duration: 0.3 }}

>

{prefix}{displayValue}{suffix}

</motion.span>

);

};

**11. Security & Validation**

**Input Validation Schemas (Zod)**

import { z } from 'zod';

// Notification settings validation schema

export const NotificationSettingsSchema = z.object({

emailAlerts: z.boolean(),

telegramAlerts: z.boolean(),

pushNotifications: z.boolean(),

digestFrequency: z.enum(['realtime', 'hourly', 'daily', 'weekly']),

minScoreThreshold: z.number().min(50).max(100),

quietHours: z.object({

enabled: z.boolean(),

start: z.string().regex(/^([01]?[0-9]|2[0-3]):[0-5][0-9]$/, 'Invalid time format'),

end: z.string().regex(/^([01]?[0-9]|2[0-3]):[0-5][0-9]$/, 'Invalid time format'),

timezone: z.string()

}).optional(),

channels: z.object({

email: z.string().email('Invalid email address'),

telegram: z.string().nullable(),

webhook: z.string().url('Invalid webhook URL').nullable()

}).optional()

});

// Security settings validation schema

export const SecuritySettingsSchema = z.object({

twoFactorEnabled: z.boolean(),

sessionTimeout: z.number().min(5).max(1440), // 5 minutes to 24 hours

deviceTrustEnabled: z.boolean(),

loginNotifications: z.boolean(),

apiKeyRotationDays: z.number().min(30).max(365)

});

// UI settings validation schema

export const UISettingsSchema = z.object({

theme: z.enum(['light', 'dark', 'auto']),

language: z.enum(['en', 'ar', 'de']),

timezone: z.string(),

showPnL: z.boolean(),

compactMode: z.boolean(),

showAnimations: z.boolean(),

autoRefresh: z.boolean(),

refreshInterval: z.number().min(10).max(300), // 10 seconds to 5 minutes

numberFormat: z.enum(['us', 'eu', 'in']),

dateFormat: z.enum(['iso', 'us', 'eu'])

});

// API settings validation schema

export const APISettingsSchema = z.object({

autoTrading: z.boolean(),

webhookUrl: z.string().url('Invalid webhook URL').nullable(),

rateLimit: z.number().min(10).max(1000),

autoTradingRules: z.object({

minScore: z.number().min(70).max(100),

maxRiskPerTrade: z.number().min(0.5).max(10),

maxDailyRisk: z.number().min(1).max(50),

allowedMarkets: z.array(z.string()).min(1, 'At least one market must be selected')

}).optional()

});

// Password change validation schema

export const ChangePasswordSchema = z.object({

currentPassword: z.string().min(1, 'Current password is required'),

newPassword: z.string()

.min(8, 'Password must be at least 8 characters')

.regex(/^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]/,

'Password must contain uppercase, lowercase, number, and special character'),

confirmPassword: z.string()

}).refine(data => data.newPassword === data.confirmPassword, {

message: "Passwords don't match",

path: ["confirmPassword"]

});

// API key generation validation schema

export const GenerateAPIKeySchema = z.object({

name: z.string().min(1, 'API key name is required').max(50, 'Name too long'),

permissions: z.array(z.enum(['read', 'write', 'admin'])).min(1, 'At least one permission required'),

expiresAt: z.string().datetime().optional()

});

// Combined settings validation

export const UserSettingsSchema = z.object({

notifications: NotificationSettingsSchema.optional(),

security: SecuritySettingsSchema.optional(),

ui: UISettingsSchema.optional(),

api: APISettingsSchema.optional()

});

// Validation hook for forms

export const useSettingsValidation = () => {

const validateNotificationSettings = (data: any) => {

try {

return { data: NotificationSettingsSchema.parse(data), errors: null };

} catch (error) {

if (error instanceof z.ZodError) {

return { data: null, errors: error.errors };

}

throw error;

}

};

const validateSecuritySettings = (data: any) => {

try {

return { data: SecuritySettingsSchema.parse(data), errors: null };

} catch (error) {

if (error instanceof z.ZodError) {

return { data: null, errors: error.errors };

}

throw error;

}

};

const validateChangePassword = (data: any) => {

try {

return { data: ChangePasswordSchema.parse(data), errors: null };

} catch (error) {

if (error instanceof z.ZodError) {

return { data: null, errors: error.errors };

}

throw error;

}

};

const validateUserSettings = (data: any) => {

try {

return { data: UserSettingsSchema.parse(data), errors: null };

} catch (error) {

if (error instanceof z.ZodError) {

return { data: null, errors: error.errors };

}

throw error;

}

};

return {

validateNotificationSettings,

validateSecuritySettings,

validateChangePassword,

validateUserSettings

};

};

**Authentication & Authorization**

// Settings access control

export const useSettingsAccess = () => {

const { user, isAuthenticated } = useAuth();

const canAccessSettings = useMemo(() => {

return isAuthenticated && user;

}, [isAuthenticated, user]);

const canModifySettings = useMemo(() => {

return canAccessSettings && user?.emailVerified;

}, [canAccessSettings, user?.emailVerified]);

const canAccessAPISettings = useMemo(() => {

return canModifySettings && user?.role !== 'viewer';

}, [canModifySettings, user?.role]);

const canEnableAutoTrading = useMemo(() => {

return canAccessAPISettings && user?.subscription?.plan !== 'free';

}, [canAccessAPISettings, user?.subscription?.plan]);

const requiresUpgrade = (feature: string): boolean => {

const premiumFeatures = ['autoTrading', 'webhooks', 'advancedSecurity'];

return premiumFeatures.includes(feature) && user?.subscription?.plan === 'free';

};

return {

canAccessSettings,

canModifySettings,

canAccessAPISettings,

canEnableAutoTrading,

requiresUpgrade

};

};

// Protected settings component wrapper

export const ProtectedSettingsComponent: React.FC<{

requiredPermission: 'view' | 'modify' | 'api' | 'autotrading';

children: ReactNode;

fallback?: ReactNode;

}> = ({ requiredPermission, children, fallback = null }) => {

const {

canAccessSettings,

canModifySettings,

canAccessAPISettings,

canEnableAutoTrading

} = useSettingsAccess();

const hasPermission = useMemo(() => {

switch (requiredPermission) {

case 'view': return canAccessSettings;

case 'modify': return canModifySettings;

case 'api': return canAccessAPISettings;

case 'autotrading': return canEnableAutoTrading;

default: return false;

}

}, [requiredPermission, canAccessSettings, canModifySettings, canAccessAPISettings, canEnableAutoTrading]);

if (!hasPermission) {

return fallback ? <>{fallback}</> : null;

}

return <>{children}</>;

};

// Settings route protection

export const SettingsRouteGuard: React.FC<{ children: ReactNode }> = ({ children }) => {

const { canAccessSettings } = useSettingsAccess();

const navigate = useNavigate();

useEffect(() => {

if (!canAccessSettings) {

navigate('/signin', { replace: true });

}

}, [canAccessSettings, navigate]);

if (!canAccessSettings) {

return (

<Layout>

<div className="min-h-screen flex items-center justify-center">

<div className="text-center">

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

<h2 className="text-xl font-semibold text-white mb-2">Access Restricted</h2>

<p className="text-slate-400 mb-4">You need to be logged in to access settings.</p>

<Button onClick={() => navigate('/signin')}>Sign In</Button>

</div>

</div>

</Layout>

);

}

return <>{children}</>;

};

**Data Sanitization & XSS Prevention**

import DOMPurify from 'dompurify';

// Input sanitization utilities

export const sanitizeInput = {

// Sanitize text input

text: (input: string): string => {

return DOMPurify.sanitize(input, { ALLOWED\_TAGS: [] });

},

// Sanitize HTML input (for rich text)

html: (input: string): string => {

return DOMPurify.sanitize(input, {

ALLOWED\_TAGS: ['b', 'i', 'u', 'strong', 'em'],

ALLOWED\_ATTR: []

});

},

// Sanitize URL input

url: (input: string): string => {

try {

const url = new URL(input);

// Only allow HTTPS URLs

if (url.protocol !== 'https:') {

throw new Error('Only HTTPS URLs are allowed');

}

return url.toString();

} catch {

return '';

}

},

// Sanitize API key input

apiKey: (input: string): string => {

// Remove any non-alphanumeric characters except hyphens and underscores

return input.replace(/[^a-zA-Z0-9\-\_]/g, '');

},

// Sanitize webhook payload

webhookPayload: (payload: any): any => {

if (typeof payload !== 'object' || payload === null) {

return {};

}

const sanitized: any = {};

for (const [key, value] of Object.entries(payload)) {

const sanitizedKey = sanitizeInput.text(key);

if (typeof value === 'string') {

sanitized[sanitizedKey] = sanitizeInput.text(value);

} else if (typeof value === 'number' || typeof value === 'boolean') {

sanitized[sanitizedKey] = value;

}

}

return sanitized;

}

};

// Sanitized input component

export const SanitizedInput: React.FC<{

type?: 'text' | 'url' | 'apikey';

value: string;

onChange: (value: string) => void;

onBlur?: () => void;

className?: string;

placeholder?: string;

}> = ({ type = 'text', value, onChange, onBlur, className, placeholder }) => {

const [localValue, setLocalValue] = useState(value);

const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {

const newValue = e.target.value;

setLocalValue(newValue);

// Real-time sanitization based on type

let sanitizedValue = newValue;

switch (type) {

case 'url':

// Allow typing, validate on blur

break;

case 'apikey':

sanitizedValue = sanitizeInput.apiKey(newValue);

break;

default:

sanitizedValue = sanitizeInput.text(newValue);

}

if (sanitizedValue !== newValue) {

setLocalValue(sanitizedValue);

}

onChange(sanitizedValue);

};

const handleBlur = () => {

// Additional sanitization on blur

let sanitizedValue = localValue;

switch (type) {

case 'url':

sanitizedValue = sanitizeInput.url(localValue);

if (sanitizedValue !== localValue) {

setLocalValue(sanitizedValue);

onChange(sanitizedValue);

}

break;

}

onBlur?.();

};

return (

<Input

value={localValue}

onChange={handleChange}

onBlur={handleBlur}

className={className}

placeholder={placeholder}

/>

);

};

**Rate Limiting & API Security**

// Client-side rate limiting

export const useRateLimit = (maxRequests: number, windowMs: number) => {

const [requests, setRequests] = useState<number[]>([]);

const isAllowed = useCallback(() => {

const now = Date.now();

const windowStart = now - windowMs;

// Filter out requests outside the current window

const recentRequests = requests.filter(timestamp => timestamp > windowStart);

// Update the requests array

setRequests(recentRequests);

return recentRequests.length < maxRequests;

}, [requests, maxRequests, windowMs]);

const recordRequest = useCallback(() => {

const now = Date.now();

setRequests(prev => [...prev, now]);

}, []);

const getRemainingRequests = useCallback(() => {

const now = Date.now();

const windowStart = now - windowMs;

const recentRequests = requests.filter(timestamp => timestamp > windowStart);

return Math.max(0, maxRequests - recentRequests.length);

}, [requests, maxRequests, windowMs]);

const getResetTime = useCallback(() => {

if (requests.length === 0) return 0;

const oldestRequest = Math.min(...requests);

return oldestRequest + windowMs;

}, [requests, windowMs]);

return {

isAllowed,

recordRequest,

getRemainingRequests,

getResetTime

};

};

// Settings API with rate limiting

export const useSettingsAPI = () => {

const rateLimit = useRateLimit(10, 60000); // 10 requests per minute

const makeRequest = async <T>(

requestFn: () => Promise<T>,

skipRateLimit = false

): Promise<T> => {

if (!skipRateLimit && !rateLimit.isAllowed()) {

const resetTime = rateLimit.getResetTime();

const waitTime = Math.ceil((resetTime - Date.now()) / 1000);

throw new Error(`Rate limit exceeded. Try again in ${waitTime} seconds.`);

}

if (!skipRateLimit) {

rateLimit.recordRequest();

}

try {

return await requestFn();

} catch (error) {

if (error instanceof Response && error.status === 429) {

throw new Error('Rate limit exceeded. Please try again later.');

}

throw error;

}

};

const updateSettings = async (settings: any) => {

return makeRequest(() => settingsAPI.updateUserSettings(settings));

};

const testNotification = async (type: string) => {

return makeRequest(() => settingsAPI.testNotification({ type } as any));

};

const generateAPIKey = async (request: any) => {

return makeRequest(() => settingsAPI.generateAPIKey(request));

};

return {

updateSettings,

testNotification,

generateAPIKey,

remainingRequests: rateLimit.getRemainingRequests()

};

};

**Sensitive Data Handling**

// Secure storage utilities

export const secureStorage = {

// Encrypt sensitive data before storing

encrypt: (data: string, key: string): string => {

// Use Web Crypto API for encryption

// This is a simplified example - use proper encryption in production

return btoa(data); // Base64 encoding as placeholder

},

// Decrypt sensitive data after retrieving

decrypt: (encryptedData: string, key: string): string => {

// Use Web Crypto API for decryption

return atob(encryptedData); // Base64 decoding as placeholder

},

// Store encrypted data

setItem: (key: string, value: any, sensitive = false): void => {

const serialized = JSON.stringify(value);

if (sensitive) {

const encrypted = secureStorage.encrypt(serialized, key);

sessionStorage.setItem(key, encrypted);

} else {

localStorage.setItem(key, serialized);

}

},

// Retrieve and decrypt data

getItem: (key: string, sensitive = false): any => {

const stored = sensitive ? sessionStorage.getItem(key) : localStorage.getItem(key);

if (!stored) return null;

try {

const decrypted = sensitive ? secureStorage.decrypt(stored, key) : stored;

return JSON.parse(decrypted);

} catch {

return null;

}

},

// Remove sensitive data

removeItem: (key: string, sensitive = false): void => {

if (sensitive) {

sessionStorage.removeItem(key);

} else {

localStorage.removeItem(key);

}

},

// Clear all sensitive data

clearSensitive: (): void => {

sessionStorage.clear();

}

};

// Secure API key display component

export const SecureAPIKeyDisplay: React.FC<{

apiKey: string;

name: string;

onRevoke: () => void;

}> = ({ apiKey, name, onRevoke }) => {

const [showFullKey, setShowFullKey] = useState(false);

const [copySuccess, setCopySuccess] = useState(false);

const maskedKey = useMemo(() => {

if (apiKey.length <= 8) return apiKey;

return `${apiKey.slice(0, 8)}${'\*'.repeat(apiKey.length - 12)}${apiKey.slice(-4)}`;

}, [apiKey]);

const copyToClipboard = async () => {

try {

await navigator.clipboard.writeText(apiKey);

setCopySuccess(true);

setTimeout(() => setCopySuccess(false), 2000);

} catch (error) {

console.error('Failed to copy API key:', error);

}

};

const handleRevoke = () => {

if (confirm(`Are you sure you want to revoke the API key "${name}"? This action cannot be undone.`)) {

onRevoke();

}

};

return (

<div className="flex items-center justify-between p-4 bg-slate-800/30 rounded-lg border border-slate-600">

<div className="flex-1">

<div className="flex items-center space-x-2 mb-2">

<Key className="h-4 w-4 text-yellow-400" />

<span className="text-white font-medium">{name}</span>

</div>

<div className="flex items-center space-x-2">

<code className="text-sm font-mono text-slate-300 bg-slate-700 px-2 py-1 rounded">

{showFullKey ? apiKey : maskedKey}

</code>

<Button

variant="ghost"

size="sm"

onClick={() => setShowFullKey(!showFullKey)}

className="text-slate-400 hover:text-white"

>

{showFullKey ? <EyeOff className="h-4 w-4" /> : <Eye className="h-4 w-4" />}

</Button>

<Button

variant="ghost"

size="sm"

onClick={copyToClipboard}

className="text-slate-400 hover:text-white"

>

{copySuccess ? <Check className="h-4 w-4" /> : <Copy className="h-4 w-4" />}

</Button>

</div>

</div>

<Button

variant="destructive"

size="sm"

onClick={handleRevoke}

className="ml-4"

>

Revoke

</Button>

</div>

);

};

**12. Environment & Configuration**

**Environment Variables**

# API Configuration

VITE\_API\_URL=https://api.kurzora.com

VITE\_API\_VERSION=v1

VITE\_WS\_URL=wss://ws.kurzora.com

# Authentication

VITE\_JWT\_ISSUER=kurzora-auth

JWT\_ACCESS\_SECRET=your-super-secret-access-key

JWT\_REFRESH\_SECRET=your-super-secret-refresh-key

JWT\_ACCESS\_EXPIRY=15m

JWT\_REFRESH\_EXPIRY=7d

# OAuth Providers

VITE\_GOOGLE\_CLIENT\_ID=your-google-client-id.googleusercontent.com

GOOGLE\_CLIENT\_SECRET=your-google-client-secret

VITE\_GITHUB\_CLIENT\_ID=your-github-client-id

GITHUB\_CLIENT\_SECRET=your-github-client-secret

# Database

DATABASE\_URL=postgresql://user:password@localhost:5432/kurzora

REDIS\_URL=redis://localhost:6379

# Email Services

SENDGRID\_API\_KEY=SG.your-sendgrid-api-key

SMTP\_HOST=smtp.sendgrid.net

SMTP\_PORT=587

SMTP\_USER=apikey

SMTP\_PASS=your-sendgrid-api-key

# Security

BCRYPT\_ROUNDS=12

RATE\_LIMIT\_WINDOW\_MS=900000

RATE\_LIMIT\_MAX\_ATTEMPTS=5

SESSION\_TIMEOUT\_MS=86400000

DEVICE\_TRUST\_DURATION\_DAYS=30

# Feature Flags

VITE\_ENABLE\_2FA=true

VITE\_ENABLE\_API\_KEYS=true

VITE\_ENABLE\_AUTO\_TRADING=true

VITE\_ENABLE\_WEBHOOKS=false

VITE\_ENABLE\_EXPORT\_DATA=true

VITE\_ENABLE\_IMPORT\_SETTINGS=false

# Monitoring & Analytics

VITE\_SENTRY\_DSN=https://your-sentry-dsn@sentry.io/project-id

SENTRY\_AUTH\_TOKEN=your-sentry-auth-token

VITE\_GA\_TRACKING\_ID=G-XXXXXXXXXX

VITE\_MIXPANEL\_TOKEN=your-mixpanel-token

# Development

VITE\_DEBUG\_MODE=false

VITE\_MOCK\_SETTINGS=false

VITE\_DISABLE\_RATE\_LIMITING=false

NODE\_ENV=production

# Security Headers

VITE\_CSP\_NONCE=true

VITE\_ENABLE\_HSTS=true

VITE\_ENABLE\_XSS\_PROTECTION=true

**Feature Flags Implementation**

// Feature flag configuration

interface FeatureFlags {

enable2FA: boolean;

enableAPIKeys: boolean;

enableAutoTrading: boolean;

enableWebhooks: boolean;

enableExportData: boolean;

enableImportSettings: boolean;

enableAdvancedSecurity: boolean;

enableRealTimeSync: boolean;

enableSettingsAnalytics: boolean;

enableBetaFeatures: boolean;

}

// Feature flag hook

const useFeatureFlags = (): FeatureFlags => {

return useMemo(() => ({

enable2FA: import.meta.env.VITE\_ENABLE\_2FA === 'true',

enableAPIKeys: import.meta.env.VITE\_ENABLE\_API\_KEYS === 'true',

enableAutoTrading: import.meta.env.VITE\_ENABLE\_AUTO\_TRADING === 'true',

enableWebhooks: import.meta.env.VITE\_ENABLE\_WEBHOOKS === 'true',

enableExportData: import.meta.env.VITE\_ENABLE\_EXPORT\_DATA === 'true',

enableImportSettings: import.meta.env.VITE\_ENABLE\_IMPORT\_SETTINGS === 'true',

enableAdvancedSecurity: import.meta.env.VITE\_ENABLE\_ADVANCED\_SECURITY === 'true',

enableRealTimeSync: import.meta.env.VITE\_ENABLE\_REAL\_TIME\_SYNC !== 'false',

enableSettingsAnalytics: import.meta.env.VITE\_ENABLE\_SETTINGS\_ANALYTICS !== 'false',

enableBetaFeatures: import.meta.env.NODE\_ENV === 'development' || import.meta.env.VITE\_ENABLE\_BETA === 'true'

}), []);

};

// Conditional rendering based on feature flags

const ConditionalFeature: React.FC<{

flag: keyof FeatureFlags;

children: React.ReactNode;

fallback?: React.ReactNode;

}> = ({ flag, children, fallback = null }) => {

const featureFlags = useFeatureFlags();

return featureFlags[flag] ? <>{children}</> : <>{fallback}</>;

};

// Usage in settings components

const SettingsPage = () => {

const featureFlags = useFeatureFlags();

return (

<Layout>

<div className="settings-container">

{/\* Always visible components \*/}

<NotificationSettingsCard />

<UISettingsCard />

{/\* Conditional components \*/}

<ConditionalFeature flag="enable2FA">

<SecuritySettingsCard />

</ConditionalFeature>

<ConditionalFeature flag="enableAPIKeys">

<APISettingsCard />

</ConditionalFeature>

<ConditionalFeature

flag="enableAdvancedSecurity"

fallback={<BasicSecurityCard />}

>

<AdvancedSecurityCard />

</ConditionalFeature>

{featureFlags.enableBetaFeatures && (

<BetaFeaturesCard />

)}

</div>

</Layout>

);

};

**Third-Party Service Configurations**

// Configuration for external services

const serviceConfigs = {

// Notification services

sendgrid: {

apiKey: import.meta.env.SENDGRID\_API\_KEY,

fromEmail: 'notifications@kurzora.com',

fromName: 'Kurzora',

templates: {

settingsChanged: 'd-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',

securityAlert: 'd-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',

welcomeEmail: 'd-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'

}

},

// Telegram bot configuration

telegram: {

botToken: import.meta.env.TELEGRAM\_BOT\_TOKEN,

baseUrl: 'https://api.telegram.org/bot',

webhookUrl: import.meta.env.TELEGRAM\_WEBHOOK\_URL,

allowedUpdates: ['message', 'callback\_query']

},

// Push notification configuration

webPush: {

vapidKeys: {

publicKey: import.meta.env.VITE\_VAPID\_PUBLIC\_KEY,

privateKey: import.meta.env.VAPID\_PRIVATE\_KEY

},

subject: 'mailto:support@kurzora.com'

},

// Security services

auth0: {

domain: import.meta.env.VITE\_AUTH0\_DOMAIN,

clientId: import.meta.env.VITE\_AUTH0\_CLIENT\_ID,

audience: import.meta.env.VITE\_AUTH0\_AUDIENCE

},

// Analytics configuration

analytics: {

googleAnalytics: {

trackingId: import.meta.env.VITE\_GA\_TRACKING\_ID,

config: {

anonymize\_ip: true,

cookie\_expires: 63072000

}

},

mixpanel: {

token: import.meta.env.VITE\_MIXPANEL\_TOKEN,

config: {

debug: process.env.NODE\_ENV === 'development',

track\_pageview: true,

persistence: 'localStorage'

}

},

// Production monitoring configuration

MONITORING\_CONFIG = {

// Error tracking

sentry: {

dsn: process.env.VITE\_SENTRY\_DSN,

environment: process.env.NODE\_ENV,

tracesSampleRate: process.env.NODE\_ENV === 'production' ? 0.1 : 1.0,

beforeSend: (event) => {

// Filter out sensitive settings data

if (event.exception) {

event.exception.values?.forEach(exception => {

if (exception.stacktrace?.frames) {

exception.stacktrace.frames = exception.stacktrace.frames.map(frame => ({

...frame,

vars: undefined // Remove variables to prevent sensitive data leaks

}));

}

});

}

return event;

}

},

// Analytics for settings usage

analytics: {

mixpanel: {

token: process.env.VITE\_MIXPANEL\_TOKEN,

trackSettingsChanges: true,

trackSecurityEvents: true,

trackPerformance: false // Disable in development

},

// Settings-specific events

events: {

SETTINGS\_PAGE\_VIEWED: 'settings\_page\_viewed',

NOTIFICATION\_SETTING\_CHANGED: 'notification\_setting\_changed',

SECURITY\_SETTING\_CHANGED: 'security\_setting\_changed',

API\_KEY\_GENERATED: 'api\_key\_generated',

TWO\_FA\_ENABLED: 'two\_fa\_enabled',

LANGUAGE\_CHANGED: 'language\_changed',

SETTINGS\_EXPORTED: 'settings\_exported',

SETTINGS\_IMPORTED: 'settings\_imported'

}

}

};

// Performance budgets for settings page

const PERFORMANCE\_BUDGETS = {

// Page load time targets

TIME\_TO\_INTERACTIVE: 3000, // 3 seconds

LARGEST\_CONTENTFUL\_PAINT: 2500, // 2.5 seconds

CUMULATIVE\_LAYOUT\_SHIFT: 0.1,

// Bundle size limits

INITIAL\_BUNDLE\_SIZE: 250000, // 250KB

SETTINGS\_CHUNK\_SIZE: 100000, // 100KB

// API response times

SETTINGS\_LOAD\_TIME: 1000, // 1 second

SETTINGS\_SAVE\_TIME: 2000, // 2 seconds

NOTIFICATION\_TEST\_TIME: 5000 // 5 seconds

};

// Settings-specific monitoring

export const monitorSettingsPerformance = () => {

// Track settings page metrics

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

list.getEntries().forEach((entry) => {

if (entry.entryType === 'measure' && entry.name.includes('settings')) {

analytics.track('settings\_performance', {

metric: entry.name,

duration: entry.duration,

timestamp: Date.now()

});

}

});

});

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

// Monitor settings save operations

const trackSettingsSave = (startTime: number, endTime: number, success: boolean) => {

const duration = endTime - startTime;

analytics.track('settings\_save\_performance', {

duration,

success,

exceeded\_budget: duration > PERFORMANCE\_BUDGETS.SETTINGS\_SAVE\_TIME

});

};

return { trackSettingsSave };

};

// Security headers for settings page

export const SECURITY\_HEADERS = {

'Content-Security-Policy': `

default-src 'self';

script-src 'self' 'unsafe-inline' 'unsafe-eval' https://apis.google.com;

style-src 'self' 'unsafe-inline' https://fonts.googleapis.com;

font-src 'self' https://fonts.gstatic.com;

img-src 'self' data: https:;

connect-src 'self' wss: https://api.kurzora.com https://ws.kurzora.com;

frame-src 'self' https://accounts.google.com;

`.replace(/\s+/g, ' ').trim(),

'X-Content-Type-Options': 'nosniff',

'X-Frame-Options': 'DENY',

'X-XSS-Protection': '1; mode=block',

'Referrer-Policy': 'strict-origin-when-cross-origin',

'Permissions-Policy': 'camera=(), microphone=(), geolocation=()'

};

// API rate limiting configuration

export const RATE\_LIMITS = {

// Settings operations

SETTINGS\_UPDATE: { requests: 10, window: 60000 }, // 10 requests per minute

NOTIFICATION\_TEST: { requests: 3, window: 60000 }, // 3 tests per minute

API\_KEY\_GENERATION: { requests: 5, window: 3600000 }, // 5 keys per hour

PASSWORD\_CHANGE: { requests: 3, window: 3600000 }, // 3 changes per hour

TWO\_FA\_ATTEMPTS: { requests: 5, window: 300000 }, // 5 attempts per 5 minutes

// Security-sensitive operations

EXPORT\_DATA: { requests: 2, window: 86400000 }, // 2 exports per day

IMPORT\_SETTINGS: { requests: 1, window: 3600000 }, // 1 import per hour

SECURITY\_LOG\_ACCESS: { requests: 10, window: 300000 } // 10 views per 5 minutes

};

// Encryption configuration for sensitive settings

export const ENCRYPTION\_CONFIG = {

ALGORITHM: 'AES-GCM',

KEY\_LENGTH: 256,

IV\_LENGTH: 12,

TAG\_LENGTH: 16,

// Fields that require encryption in storage

ENCRYPTED\_FIELDS: [

'api\_keys',

'two\_fa\_secret',

'backup\_codes',

'webhook\_urls',

'broker\_credentials'

]

};

**13. Cross-Screen Data Flow**

**Global Settings Synchronization**

// Global settings event system for cross-application synchronization

export const SettingsEventBus = {

// Event types that affect other parts of the application

events: {

LANGUAGE\_CHANGED: 'settings:language\_changed',

THEME\_CHANGED: 'settings:theme\_changed',

TIMEZONE\_CHANGED: 'settings:timezone\_changed',

NOTIFICATIONS\_UPDATED: 'settings:notifications\_updated',

API\_SETTINGS\_UPDATED: 'settings:api\_updated',

SECURITY\_SETTINGS\_UPDATED: 'settings:security\_updated',

AUTO\_REFRESH\_TOGGLED: 'settings:auto\_refresh\_toggled',

DISPLAY\_PREFERENCES\_UPDATED: 'settings:display\_updated'

},

// Event emitter

emit: (eventType: string, data: any) => {

document.dispatchEvent(new CustomEvent(eventType, { detail: data }));

// Also propagate through WebSocket for multi-tab sync

if (window.kurzora?.websocket?.connected) {

window.kurzora.websocket.send(JSON.stringify({

type: 'settings\_sync',

event: eventType,

data,

timestamp: Date.now()

}));

}

},

// Event listener registration

subscribe: (eventType: string, handler: (data: any) => void) => {

const listener = (event: CustomEvent) => handler(event.detail);

document.addEventListener(eventType, listener);

return () => document.removeEventListener(eventType, listener);

}

};

// Global settings synchronization hook

export const useGlobalSettingsSync = () => {

const settingsStore = useSettingsStore();

const signalsStore = useSignalsStore();

const dashboardStore = useDashboardStore();

const authStore = useAuthStore();

useEffect(() => {

// Language change affects all text and date formatting

const handleLanguageChange = (data: { language: string; timezone: string }) => {

// Update global language context

useLanguageStore.getState().setLanguage(data.language);

useLanguageStore.getState().setTimezone(data.timezone);

// Apply RTL layout for Arabic

document.documentElement.dir = data.language === 'ar' ? 'rtl' : 'ltr';

document.documentElement.lang = data.language;

// Update date formatting across all stores

signalsStore.refreshDateFormats(data.timezone);

dashboardStore.refreshDateFormats(data.timezone);

// Invalidate cached content that depends on language

queryClient.invalidateQueries(['translations']);

queryClient.invalidateQueries(['user-content']);

};

// Theme change affects UI across all pages

const handleThemeChange = (data: { theme: 'light' | 'dark' | 'auto' }) => {

document.documentElement.setAttribute('data-theme', data.theme);

// Update chart themes in signals and dashboard

signalsStore.updateChartTheme(data.theme);

dashboardStore.updateChartTheme(data.theme);

// Persist theme preference

localStorage.setItem('kurzora-theme', data.theme);

};

// Notification settings affect alert behavior

const handleNotificationsUpdate = (data: NotificationSettings) => {

// Update notification service configuration

notificationService.updateSettings(data);

// Update alerts display preferences in dashboard

dashboardStore.updateAlertSettings({

showNotifications: data.pushNotifications,

soundEnabled: data.soundEnabled,

quietHours: data.quietHours

});

// Update signal alert thresholds

signalsStore.updateAlertThreshold(data.minScoreThreshold);

};

// Auto-refresh setting affects data polling

const handleAutoRefreshToggle = (data: { enabled: boolean; interval: number }) => {

if (data.enabled) {

// Start auto-refresh for signals and dashboard

signalsStore.startAutoRefresh(data.interval);

dashboardStore.startAutoRefresh(data.interval);

} else {

// Stop auto-refresh

signalsStore.stopAutoRefresh();

dashboardStore.stopAutoRefresh();

}

};

// API settings affect data fetching

const handleAPISettingsUpdate = (data: APISettings) => {

// Update API client configuration

apiClient.updateRateLimit(data.rateLimit);

// Update auto-trading status in signals

if (data.autoTrading) {

signalsStore.enableAutoTrading(data.autoTradingRules);

} else {

signalsStore.disableAutoTrading();

}

// Update webhook configurations

webhookService.updateEndpoints(data.webhookUrl);

};

// Security settings affect authentication flow

const handleSecurityUpdate = (data: SecuritySettings) => {

// Update session timeout

authStore.updateSessionTimeout(data.sessionTimeout);

// Update 2FA status

authStore.update2FAStatus(data.twoFactorEnabled);

// Update trusted device settings

if (!data.deviceTrustEnabled) {

authStore.clearTrustedDevices();

}

};

// Display preferences affect UI components

const handleDisplayUpdate = (data: UISettings) => {

// Update number formatting across all components

const formatter = new Intl.NumberFormat(data.numberFormat === 'us' ? 'en-US' :

data.numberFormat === 'eu' ? 'de-DE' : 'en-IN');

// Update stores with new formatter

signalsStore.updateNumberFormatter(formatter);

dashboardStore.updateNumberFormatter(formatter);

// Update compact mode for mobile displays

if (data.compactMode) {

document.body.classList.add('compact-mode');

} else {

document.body.classList.remove('compact-mode');

}

// Update animation preferences

if (!data.showAnimations) {

document.body.classList.add('reduced-motion');

} else {

document.body.classList.remove('reduced-motion');

}

};

// Register event listeners

const unsubscribers = [

SettingsEventBus.subscribe(SettingsEventBus.events.LANGUAGE\_CHANGED, handleLanguageChange),

SettingsEventBus.subscribe(SettingsEventBus.events.THEME\_CHANGED, handleThemeChange),

SettingsEventBus.subscribe(SettingsEventBus.events.NOTIFICATIONS\_UPDATED, handleNotificationsUpdate),

SettingsEventBus.subscribe(SettingsEventBus.events.AUTO\_REFRESH\_TOGGLED, handleAutoRefreshToggle),

SettingsEventBus.subscribe(SettingsEventBus.events.API\_SETTINGS\_UPDATED, handleAPISettingsUpdate),

SettingsEventBus.subscribe(SettingsEventBus.events.SECURITY\_SETTINGS\_UPDATED, handleSecurityUpdate),

SettingsEventBus.subscribe(SettingsEventBus.events.DISPLAY\_PREFERENCES\_UPDATED, handleDisplayUpdate)

];

return () => {

unsubscribers.forEach(unsub => unsub());

};

}, []);

};

Real-Time Settings Synchronization

// Multi-tab and multi-device settings synchronization

export const useRealtimeSettingsSync = () => {

const settingsStore = useSettingsStore();

const { user } = useAuth();

useEffect(() => {

if (!user) return;

// WebSocket connection for real-time sync

const wsUrl = `${import.meta.env.VITE\_WS\_URL}/settings?userId=${user.id}`;

const ws = new WebSocket(wsUrl);

ws.onopen = () => {

console.log('Settings sync WebSocket connected');

};

ws.onmessage = (event) => {

try {

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

switch (message.type) {

case 'SETTINGS\_UPDATED':

// Another tab/device updated settings

handleRemoteSettingsUpdate(message.data);

break;

case 'SETTINGS\_CONFLICT':

// Concurrent modifications detected

handleSettingsConflict(message.data);

break;

case 'SECURITY\_ALERT':

// Security-related settings changed

handleSecurityAlert(message.data);

break;

}

} catch (error) {

console.error('Error processing settings sync message:', error);

}

};

const handleRemoteSettingsUpdate = (remoteSettings: any) => {

const currentSettings = settingsStore.getAllSettings();

const conflicts = detectSettingsConflicts(currentSettings, remoteSettings);

if (conflicts.length === 0) {

// No conflicts, apply remote changes

settingsStore.mergeSettings(remoteSettings);

toast.info('Settings updated from another device');

} else {

// Show conflict resolution UI

settingsStore.showConflictResolution(conflicts);

}

};

const handleSettingsConflict = (conflictData: any) => {

// Show modal for user to resolve conflicts

settingsStore.openConflictResolutionModal({

localChanges: conflictData.local,

remoteChanges: conflictData.remote,

conflictFields: conflictData.conflicts

});

};

const handleSecurityAlert = (alertData: any) => {

if (alertData.severity === 'high') {

// Force re-authentication for high-security changes

authStore.requireReauthentication();

}

toast.warning(`Security settings changed: ${alertData.message}`);

};

return () => {

ws.close();

};

}, [user]);

};

// Settings conflict detection and resolution

const detectSettingsConflicts = (local: any, remote: any) => {

const conflicts: Array<{

field: string;

localValue: any;

remoteValue: any;

lastModified: { local: Date; remote: Date };

}> = [];

const compareSettings = (localObj: any, remoteObj: any, path = '') => {

Object.keys(localObj).forEach(key => {

const currentPath = path ? `${path}.${key}` : key;

const localValue = localObj[key];

const remoteValue = remoteObj[key];

if (typeof localValue === 'object' && typeof remoteValue === 'object') {

compareSettings(localValue, remoteValue, currentPath);

} else if (localValue !== remoteValue) {

conflicts.push({

field: currentPath,

localValue,

remoteValue,

lastModified: {

local: new Date(localObj.\_lastModified || 0),

remote: new Date(remoteObj.\_lastModified || 0)

}

});

}

});

};

compareSettings(local, remote);

return conflicts;

};

Cross-Page State Dependencies

// Settings dependencies that affect other pages

export const SettingsDependencyManager = {

// Map of settings to affected stores/components

dependencies: {

'notifications.minScoreThreshold': ['signals', 'dashboard'],

'ui.autoRefresh': ['signals', 'dashboard', 'trades'],

'ui.showPnL': ['dashboard', 'trades', 'portfolio'],

'ui.compactMode': ['signals', 'dashboard', 'trades'],

'ui.theme': ['global'],

'ui.language': ['global'],

'security.sessionTimeout': ['auth'],

'api.autoTrading': ['signals', 'trades'],

'api.rateLimit': ['api-client']

},

// Propagate setting changes to dependent systems

propagateChange: (settingPath: string, newValue: any, oldValue: any) => {

const affectedSystems = SettingsDependencyManager.dependencies[settingPath] || [];

affectedSystems.forEach(system => {

switch (system) {

case 'signals':

useSignalsStore.getState().onSettingChanged(settingPath, newValue, oldValue);

break;

case 'dashboard':

useDashboardStore.getState().onSettingChanged(settingPath, newValue, oldValue);

break;

case 'trades':

useTradesStore.getState().onSettingChanged(settingPath, newValue, oldValue);

break;

case 'auth':

useAuthStore.getState().onSettingChanged(settingPath, newValue, oldValue);

break;

case 'api-client':

updateAPIClientConfig(settingPath, newValue);

break;

case 'global':

updateGlobalConfig(settingPath, newValue);

break;

}

});

// Invalidate affected React Query caches

const affectedQueries = getCacheKeysForSetting(settingPath);

affectedQueries.forEach(queryKey => {

queryClient.invalidateQueries(queryKey);

});

},

// Get all settings that affect a specific system

getSystemDependencies: (systemName: string) => {

return Object.entries(SettingsDependencyManager.dependencies)

.filter(([\_, systems]) => systems.includes(systemName))

.map(([settingPath]) => settingPath);

}

};

// Cache invalidation strategy for settings changes

const getCacheKeysForSetting = (settingPath: string): string[][] => {

const cacheInvalidationMap: Record<string, string[][]> = {

'ui.language': [['translations'], ['user-content'], ['signal-descriptions']],

'ui.timezone': [['dashboard-data'], ['trade-history'], ['signal-history']],

'notifications.minScoreThreshold': [['dashboard-alerts'], ['active-signals']],

'api.autoTrading': [['trading-status'], ['auto-trades']],

'ui.showPnL': [['portfolio-data'], ['trade-performance']]

};

return cacheInvalidationMap[settingPath] || [];

};

// Settings-aware component HOC for automatic re-rendering

export const withSettingsSync = <P extends object>(

Component: React.ComponentType<P>,

watchedSettings: string[]

) => {

return React.memo((props: P) => {

const [settingsVersion, setSettingsVersion] = useState(0);

useEffect(() => {

const unsubscribers = watchedSettings.map(settingPath =>

SettingsEventBus.subscribe(`settings:${settingPath}\_changed`, () => {

setSettingsVersion(prev => prev + 1);

})

);

return () => {

unsubscribers.forEach(unsub => unsub());

};

}, []);

return <Component {...props} key={settingsVersion} />;

});

};

// Usage example for components that depend on settings

const SignalsPageWithSettingsSync = withSettingsSync(SignalsPage, [

'notifications.minScoreThreshold',

'ui.autoRefresh',

'ui.compactMode'

]);