TradeSwitch App

Guards TypeScript Code

Generated: 11/25/2025, 6:40:40 PM

Total Files: 3

Table of Contents

Ø=ÜÁ guards

Ø=ÜÄ guards\auth-guard-guard.ts

```typescript
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from '../features/auth/service/authService';
import { inject } from '@angular/core';
import { catchError, from, map, of, switchMap, take, tap } from 'rxjs';
import { Store } from '@ngrx/store';
import { getAuth } from 'firebase/auth';
import { setUserData } from '../features/auth/store/user.actions';
import { ReasonsService } from '../shared/services/reasons.service';

/**
 * Authentication guard that protects routes requiring user authentication.
 *
 * This guard checks if a user is authenticated and verifies their status.
 * It prevents banned users from accessing protected routes and redirects
 * unauthenticated users to the login page.
 *
 * Features:
 * - Verifies Firebase authentication state
 * - Checks user status (banned users are blocked)
 * - Fetches and dispatches user data to NgRx store
 * - Shows ban reason if user is banned
 * - Redirects to login if not authenticated
 *
 * Flow:
 * 1. Check if user is authenticated
 * 2. Get current Firebase user
 * 3. Fetch user data from Firestore
 * 4. Check if user is banned (show reason and redirect)
 * 5. Dispatch user data to store
 * 6. Allow access if all checks pass
 *
 * Relations:
 * - AuthService: Checks authentication and fetches user data
 * - ReasonsService: Gets ban reason for banned users
 * - Store (NgRx): Dispatches user data
 * - Router: Handles navigation redirects
 *
 * @guard
 * @function authGuard
 */
export const authGuard: CanActivateFn = (route, state) => {
  const router = inject(Router);
  const authService = inject(AuthService);
  const store = inject(Store);
  const reasonsService = inject(ReasonsService);

  return authService.isAuthenticated().pipe(
    take(1),
    switchMap((isAuth) => {
      if (!isAuth) {
        router.navigate(['/login']);
        return of(false);
      }
      const user = getAuth().currentUser;
      if (!user) {
        router.navigate(['/login']);
        return of(false);
      }

      return from(authService.getUserData(user.uid)).pipe(
        switchMap((userData) => {
          if (userData.status === 'banned') {
            return from(reasonsService.getOpenLatestReason(user.uid)).pipe(
              tap((reason) => {
```

```
                const message = reason?.reason
                  ? `You are banned: ${reason.reason}`
                  : 'You are banned, call support';
                alert(message);
                router.navigate(['/login']);
              }),
              map(() => false)
            );
          }
          store.dispatch(setUserData({ user: userData }));
          return of(true);
        }),
        catchError(() => {
          router.navigate(['/login']);
          return of(false);
        })
      );
    })
  );
};

```

Ø=ÜÄ guards\plan-limitations.guard.ts

```
import { Injectable, inject } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { Store } from '@ngrx/store';
import { selectUser } from '../features/auth/store/user.selectios';
import { SubscriptionService } from '../shared/services/subscription-service';
import { PlanService } from '../shared/services/planService';
import { UserStatus } from '../features/overview/models/overview';
import { Observable, of, switchMap, catchError } from 'rxjs';
import { Subscription } from '../shared/services/subscription-service';
import { AppContextService } from '../shared/context';

export interface PlanLimitations {
  maxAccounts: number;
  maxStrategies: number;
  planName: string;
  isActive: boolean;
  isBanned: boolean;
  isCancelled: boolean;
  needsSubscription: boolean;
}

export interface LimitationCheck {
  canCreate: boolean;
  reason?: string;
  showUpgradeModal: boolean;
  upgradeMessage: string;
  showBlockedModal: boolean;
  blockedMessage: string;
}

export interface ModalData {
  showModal: boolean;
  modalType: 'upgrade' | 'blocked';
  title: string;
  message: string;
  primaryButtonText: string;
  secondaryButtonText?: string;
  onPrimaryAction: () => void;
  onSecondaryAction?: () => void;
}

/**
```

```
 * Guard and service for checking user plan limitations and feature access.
 *
 * This guard/service provides comprehensive plan limitation checking for features
 * like account creation, strategy creation, and report access. It validates
 * subscription status, plan limits, and provides modal data for blocked features.
 *
 * Features:
 * - Route guard for plan-limited features
 * - Check account creation limits
 * - Check strategy creation limits
 * - Check report access
 * - Validate subscription status (active, banned, cancelled)
 * - Generate modal data for upgrade/blocked scenarios
 * - Integration with AppContextService for plan data
 *
 * Plan Status Validation:
 * - Active: User has valid subscription
 * - Banned: User account is banned
 * - Cancelled: Subscription cancelled
 * - Needs Subscription: User needs to purchase a plan
 *
 * Relations:
 * - SubscriptionService: Gets user subscription data
 * - PlanService: Gets plan details and limits
 * - AppContextService: Accesses cached plan data
 * - Store (NgRx): Gets current user
 * - Router: Navigation for upgrade flows
 *
 * @guard
 * @service
 * @injectable
 */
@Injectable({
  providedIn: 'root'
})
export class PlanLimitationsGuard implements CanActivate {
  private subscriptionService = inject(SubscriptionService);
  private planService = inject(PlanService);
  private store = inject(Store);
  private router = inject(Router);
  private appContext = inject(AppContextService);

  canActivate(): Observable<boolean> {
    return this.store.select(selectUser).pipe(
      switchMap(async (userState) => {
        const user = userState?.user;
        if (!user?.id) {
          return false;
        }
        const limitations = await this.checkUserLimitations(user.id);
        return limitations.isActive && !limitations.needsSubscription;
      }),
      catchError(() => of(false))
    );
  }

  /**
   * Check user's plan limitations and return detailed information
   */
  async checkUserLimitations(userId: string): Promise<PlanLimitations> {
    try {
      // Usar primero el contexto global
      const ctxPlan = this.appContext.userPlan();
      if (ctxPlan) {
        const isBanned = (ctxPlan as any).status === UserStatus.BANNED || ctxPlan.isActive
=== false;
        const isCancelled = (ctxPlan as any).status === UserStatus.CANCELLED &&
ctxPlan.planName === 'Free';
        const isActive = ctxPlan.isActive && !isBanned;
        return {
          maxAccounts: ctxPlan.maxAccounts,
          maxStrategies: ctxPlan.maxStrategies,
```

```
          planName: ctxPlan.planName,
          isActive,
          isBanned,
          isCancelled,
          needsSubscription: !isActive && !isCancelled && !isBanned
        };
      }

      // Fallback: obtener la última suscripción y plan (no debería ocurrir si el contexto está activo)
      const latestSubscription: Subscription | null = await this.subscriptionService.getUserLatestSubscription(userId);
      if (!latestSubscription) {
        return {
          maxAccounts: 0,
          maxStrategies: 0,
          planName: 'No Plan',
          isActive: false,
          isBanned: false,
          isCancelled: false,
          needsSubscription: true
        };
      }

      const isBanned = latestSubscription.status === UserStatus.BANNED;
      const isCancelled = latestSubscription.status === UserStatus.CANCELLED;
      const isActive = latestSubscription.status === UserStatus.PURCHASED ||
                       latestSubscription.status === UserStatus.CREATED ||
                       latestSubscription.status === UserStatus.PROCESSING ||
                       latestSubscription.status === UserStatus.ACTIVE;

      if (isBanned || isCancelled || !isActive) {
        return {
          maxAccounts: 0,
          maxStrategies: 0,
          planName: 'Inactive Plan',
          isActive: false,
          isBanned,
          isCancelled,
          needsSubscription: true
        };
      }

      // Get plan details from Firebase
      const plan = await this.planService.getPlanById(latestSubscription.planId);

      if (!plan) {
        return {
          maxAccounts: 0,
          maxStrategies: 0,
          planName: 'Unknown Plan',
          isActive: false,
          isBanned: false,
          isCancelled: false,
          needsSubscription: true
        };
      }

      // Extract limitations from plan
      const maxAccounts = plan.tradingAccounts || 1;
      const maxStrategies = plan.strategies || 1;

      return {
        maxAccounts,
        maxStrategies,
        planName: plan.name,
        isActive: true,
        isBanned: false,
        isCancelled: false,
        needsSubscription: false
      };

```

```
    } catch (error) {
      console.error('Error checking user limitations:', error);
      return {
        maxAccounts: 0,
        maxStrategies: 0,
        planName: 'Error',
        isActive: false,
        isBanned: false,
        isCancelled: false,
        needsSubscription: true
      };
    }
  }

  /**
   * Check if user can create accounts
   */
  async checkAccountCreation(userId: string, currentAccountCount: number): Promise<limitationCheck> {
    const limitations = await this.checkUserLimitations(userId);

    // If user needs subscription or is banned/cancelled
    if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
      return {
        canCreate: false,
        showBlockedModal: true,
        blockedMessage: this.getBlockedMessage(limitations),
        showUpgradeModal: false,
        upgradeMessage: ''
      };
    }

    // Check if user has reached account limit
    if (currentAccountCount >= limitations.maxAccounts) {
      return {
        canCreate: false,
        showUpgradeModal: true,
        upgradeMessage: `You've reached the account limit for your ${limitations.planName} plan. Move to a higher plan and keep growing your account.`,
        showBlockedModal: false,
        blockedMessage: ''
      };
    }

    return {
      canCreate: true,
      showUpgradeModal: false,
      upgradeMessage: '',
      showBlockedModal: false,
      blockedMessage: ''
    };
  }

  /**
   * Check if user can create strategies
   */
  async checkStrategyCreation(userId: string, currentStrategyCount: number): Promise<limitationCheck> {
    const limitations = await this.checkUserLimitations(userId);

    // If user needs subscription or is banned/cancelled
    if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
      return {
        canCreate: false,
        showBlockedModal: true,
        blockedMessage: this.getBlockedMessage(limitations),
        showUpgradeModal: false,
        upgradeMessage: ''
      };
    }

    // Check if user has reached strategy limit
    if (currentStrategyCount >= limitations.maxStrategies) {
```

```
      return {
        canCreate: false,
        showUpgradeModal: true,
        upgradeMessage: `You've reached the strategy limit for your ${limitations.planName} plan. Move to a higher plan and keep growing your account.`,
        showBlockedModal: false,
        blockedMessage: ''
      };
    }

    return {
      canCreate: true,
      showUpgradeModal: false,
      upgradeMessage: '',
      showBlockedModal: false,
      blockedMessage: ''
    };
  }

  /**
   * Get appropriate blocked message based on user status
   */
  private getBlockedMessage(limitations: PlanLimitations): string {
    if (limitations.isBanned) {
      return 'Your account has been banned. Please contact support for assistance.';
    }

    if (limitations.isCancelled) {
      return 'Your subscription has been cancelled. Please purchase a plan to access this functionality.';
    }

    if (limitations.needsSubscription) {
      return 'You need to purchase a plan to access this functionality.';
    }

    return 'Access denied. Please contact support for assistance.';
  }

  /**
   * Navigate to account page for plan management
   */
  navigateToAccount(): void {
    this.router.navigate(['/account']);
  }

  /**
   * Navigate to signup page for new users
   */
  navigateToSignup(): void {
    this.router.navigate(['/signup']);
  }

  /**
   * Check if user can access a specific feature and return modal data if blocked
   */
  async checkFeatureAccess(
    userId: string,
    feature: 'accounts' | 'strategies' | 'reports',
    currentCount?: number
  ): Promise<{ canAccess: boolean; modalData?: ModalData }> {
    try {
      const limitations = await this.checkUserLimitations(userId);

      // If user needs subscription or is banned/cancelled
      if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
        return {
          canAccess: false,
          modalData: {
            showModal: true,
            modalType: 'blocked',
            title: 'Access Restricted',
```

```
            message: this.getBlockedMessage(limitations),
            primaryButtonText: 'Go to Account',
            onPrimaryAction: () => this.navigateToAccount()
          }
        };
      }

      // Check specific feature limits
      let maxAllowed = 0;
      let featureName = '';

      switch (feature) {
        case 'accounts':
          maxAllowed = limitations.maxAccounts;
          featureName = 'trading accounts';
          break;
        case 'strategies':
          maxAllowed = limitations.maxStrategies;
          featureName = 'strategies';
          break;
        case 'reports':
          // Reports don't have a count limit, but need active subscription
          return { canAccess: true };
      }

      // If user has reached the limit
      if (currentCount !== undefined && currentCount >= maxAllowed) {
        // Check if user is on the maximum plan (Pro plan with max limits)
        const isProPlanWithMaxLimits = limitations.planName.toLowerCase().includes('pro') &&
                                       ((feature === 'strategies' && maxAllowed === 8) ||
                                        (feature === 'accounts' && maxAllowed === 10));

        // If user is already on the maximum plan, don't show upgrade modal
        if (isProPlanWithMaxLimits) {
          return {
            canAccess: false
          };
        }

        // For other plans, show upgrade modal
        return {
          canAccess: false,
          modalData: {
            showModal: true,
            modalType: 'upgrade',
            title: 'Upgrade Required',
            message: `You've reached the ${featureName} limit for your
${limitations.planName} plan. Move to a higher plan and keep growing your account.`,
            primaryButtonText: 'Upgrade Plan',
            secondaryButtonText: 'Cancel',
            onPrimaryAction: () => this.navigateToAccount(),
            onSecondaryAction: () => {}
          }
        };
      }

      return { canAccess: true };

    } catch (error) {
      console.error('Error checking feature access:', error);
      return {
        canAccess: false,
        modalData: {
          showModal: true,
          modalType: 'blocked',
          title: 'Access Restricted',
          message: 'An error occurred while checking your access. Please try again.',
          primaryButtonText: 'Go to Account',
          onPrimaryAction: () => this.navigateToAccount()
        }
      };
```

```
    }
  }

  /**
   * Check if user can create accounts and return modal data if blocked
   */
  async checkAccountCreationWithModal(userId: string, currentAccountCount: number): Promise<{ canCreate: boolean; modalData?: ModalData }> {
    const result = await this.checkFeatureAccess(userId, 'accounts', currentAccountCount);
    return {
      canCreate: result.canAccess,
      modalData: result.modalData
    };
  }

  /**
   * Check if user can create strategies and return modal data if blocked
   */
  async checkStrategyCreationWithModal(userId: string, currentStrategyCount: number): Promise<{ canCreate: boolean; modalData?: ModalData }> {
    const result = await this.checkFeatureAccess(userId, 'strategies', currentStrategyCount);
    return {
      canCreate: result.canAccess,
      modalData: result.modalData
    };
  }

  /**
   * Check if user can access reports and return modal data if blocked
   */
  async checkReportAccessWithModal(userId: string): Promise<{ canAccess: boolean; modalData?: ModalData }> {
    return this.checkFeatureAccess(userId, 'reports');
  }
}

```

Ø=ÜÄ guards\redirect-guard.guard.ts

```
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from '../features/auth/service/authService';
import { inject } from '@angular/core';
import { catchError, from, map, of, switchMap, take, tap } from 'rxjs';
import { Store } from '@ngrx/store';
import { getAuth } from 'firebase/auth';
import { setUserData } from '../features/auth/store/user.actions';

/**
 * Redirect guard that handles post-authentication routing.
 *
 * This guard is used after successful authentication to redirect users
 * to the appropriate page based on their role. Admins are redirected to
 * the overview page, while regular users are redirected to the strategy page.
 *
 * Features:
 * - Verifies authentication state
 * - Checks user status (banned users are blocked)
 * - Role-based redirection (admin !’ overview, user !’ strategy)
 * - Dispatches user data to NgRx store
 * - Shows ban alert for banned users
 *
 * Redirect Logic:
 * - Admin users !’ /overview
 * - Regular users !’ /strategy
 * - Banned users !’ /login (with alert)
 * - Unauthenticated !’ /login
 *
 * Relations:
 * - AuthService: Checks authentication and fetches user data
```

```
 * - Store (NgRx): Dispatches user data
 * - Router: Handles navigation redirects
 *
 * @guard
 * @function redirectGuard
 */
export const redirectGuard: CanActivateFn = (route, state) => {
  const router = inject(Router);
  const authService = inject(AuthService);
  const store = inject(Store);

  return authService.isAuthenticated().pipe(
    take(1),
    switchMap((isAuth) => {
      if (!isAuth) {
        router.navigate(['/login']);
        return of(false);
      }

      const user = getAuth().currentUser;
      if (!user) {
        router.navigate(['/login']);
        return of(false);
      }

      return from(authService.getUserData(user.uid)).pipe(
        tap((userData) => {
          if (userData.status === 'banned') {
            alert('You are banned, call support');
            router.navigate(['/login']);
            throw new Error('User banned');
          }
          store.dispatch(setUserData({ user: userData }));

          // Redirección inteligente según tipo de usuario
          if (userData.isAdmin) {
            router.navigate(['/overview']);
          } else {
            router.navigate(['/strategy']);
          }
        }),
        map(() => true),
        catchError(() => {
          router.navigate(['/login']);
          return of(false);
        })
      );
    })
  );
};

```