

TradeSwitch App

Guards TypeScript Code

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

Total Files: 3

Table of Contents

1. guards (3 files)

Ø=ÜÁ guards

Ø=ÜÄ guards\auth-guard-guard.ts

```
1 import { CanActivateFn, Router } from '@angular/router';
2 import { AuthService } from '../features/auth/service/authService';
3 import { inject } from '@angular/core';
4 import { catchError, from, map, of, switchMap, take, tap } from 'rxjs';
5 import { Store } from '@ngrx/store';
6 import { getAuth } from 'firebase/auth';
7 import { setUserData } from '../features/auth/store/user.actions';
8 import { ReasonsService } from '../shared/services/reasons.service';
9
10 /**
11  * Authentication guard that protects routes requiring user authentication.
12  *
13  * This guard checks if a user is authenticated and verifies their status.
14  * It prevents banned users from accessing protected routes and redirects
15  * unauthenticated users to the login page.
16  *
17  * Features:
18  * - Verifies Firebase authentication state
19  * - Checks user status (banned users are blocked)
20  * - Fetches and dispatches user data to NgRx store
21  * - Shows ban reason if user is banned
22  * - Redirects to login if not authenticated
23  *
24  * Flow:
25  * 1. Check if user is authenticated
26  * 2. Get current Firebase user
27  * 3. Fetch user data from Firestore
28  * 4. Check if user is banned (show reason and redirect)
29  * 5. Dispatch user data to store
30  * 6. Allow access if all checks pass
31  *
32  * Relations:
33  * - AuthService: Checks authentication and fetches user data
34  * - ReasonsService: Gets ban reason for banned users
35  * - Store (NgRx): Dispatches user data
36  * - Router: Handles navigation redirects
37  *
38  * @guard
39  * @function authGuard
40  */
41 export const authGuard: CanActivateFn = (route, state) => {
42   const router = inject(Router);
43   const authService = inject(AuthService);
44   const store = inject(Store);
45   const reasonsService = inject(ReasonsService);
46
47   return authService.isAuthenticated().pipe(
48     take(1),
49     switchMap((isAuth) => {
50       if (!isAuth) {
51         router.navigate(['/login']);
52         return of(false);
53       }
54       const user = getAuth().currentUser;
55       if (!user) {
56         router.navigate(['/login']);
57         return of(false);
58       }
59
60       return from(authService.getUserData(user.uid)).pipe(
61         switchMap((userData) => {
62           if (userData.status === 'banned') {
63             return from(reasonsService.getOpenLatestReason(user.uid)).pipe(
64               tap((reason) => {
```

```

65         const message = reason?.reason
66         ? `You are banned: ${reason.reason}`
67         : 'You are banned, call support';
68         alert(message);
69         router.navigate(['/login']);
70     }},
71     map(() => false)
72 );
73 }
74 store.dispatch(setUserData({ user: userData }));
75 return of(true);
76 }},
77 catchError(() => {
78     router.navigate(['/login']);
79     return of(false);
80 })
81 );
82 })
83 );
84 };
85

```

guards\plan-limitations.guard.ts

```

1  import { Injectable, inject } from '@angular/core';
2  import { CanActivate, Router } from '@angular/router';
3  import { Store } from '@ngrx/store';
4  import { selectUser } from '../features/auth/store/user.selectios';
5  import { SubscriptionService } from '../shared/services/subscription-service';
6  import { PlanService } from '../shared/services/planService';
7  import { UserStatus } from '../features/overview/models/overview';
8  import { Observable, of, switchMap, catchError } from 'rxjs';
9  import { Subscription } from '../shared/services/subscription-service';
10 import { AppContextService } from '../shared/context';
11
12 export interface PlanLimitations {
13     maxAccounts: number;
14     maxStrategies: number;
15     planName: string;
16     isActive: boolean;
17     isBanned: boolean;
18     isCancelled: boolean;
19     needsSubscription: boolean;
20 }
21
22 export interface LimitationCheck {
23     canCreate: boolean;
24     reason?: string;
25     showUpgradeModal: boolean;
26     upgradeMessage: string;
27     showBlockedModal: boolean;
28     blockedMessage: string;
29 }
30
31 export interface ModalData {
32     showModal: boolean;
33     modalType: 'upgrade' | 'blocked';
34     title: string;
35     message: string;
36     primaryButtonText: string;
37     secondaryButtonText?: string;
38     onPrimaryAction: () => void;
39     onSecondaryAction?: () => void;
40 }
41
42 /**

```

```

43 * Guard and service for checking user plan limitations and feature access.
44 *
45 * This guard/service provides comprehensive plan limitation checking for features
46 * like account creation, strategy creation, and report access. It validates
47 * subscription status, plan limits, and provides modal data for blocked features.
48 *
49 * Features:
50 * - Route guard for plan-limited features
51 * - Check account creation limits
52 * - Check strategy creation limits
53 * - Check report access
54 * - Validate subscription status (active, banned, cancelled)
55 * - Generate modal data for upgrade/blocked scenarios
56 * - Integration with AppContextService for plan data
57 *
58 * Plan Status Validation:
59 * - Active: User has valid subscription
60 * - Banned: User account is banned
61 * - Cancelled: Subscription cancelled
62 * - Needs Subscription: User needs to purchase a plan
63 *
64 * Relations:
65 * - SubscriptionService: Gets user subscription data
66 * - PlanService: Gets plan details and limits
67 * - AppContextService: Accesses cached plan data
68 * - Store (NgRx): Gets current user
69 * - Router: Navigation for upgrade flows
70 *
71 * @guard
72 * @service
73 * @injectable
74 */
75 @Injectable({
76   providedIn: 'root'
77 })
78 export class PlanLimitationsGuard implements CanActivate {
79   private subscriptionService = inject(SubscriptionService);
80   private planService = inject(PlanService);
81   private store = inject(Store);
82   private router = inject(Router);
83   private appContext = inject(AppContextService);
84
85   canActivate(): Observable<boolean> {
86     return this.store.select(selectUser).pipe(
87       switchMap(async (userState) => {
88         const user = userState?.user;
89         if (!user?.id) {
90           return false;
91         }
92         const limitations = await this.checkUserLimitations(user.id);
93         return limitations.isActive && !limitations.needsSubscription;
94       }),
95       catchError(() => of(false))
96     );
97   }
98
99   /**
100    * Check user's plan limitations and return detailed information
101    */
102   async checkUserLimitations(userId: string): Promise<PlanLimitations> {
103     try {
104       // Usar primero el contexto global
105       const ctxPlan = this.appContext.userPlan();
106       if (ctxPlan) {
107         const isBanned = (ctxPlan as any).status === UserStatus.BANNED || ctxPlan.isActive
108         === false;
109         const isCancelled = (ctxPlan as any).status === UserStatus.CANCELLED &&
110         ctxPlan.planName === 'Free';
111         const isActive = ctxPlan.isActive && !isBanned;
112         return {
113           maxAccounts: ctxPlan.maxAccounts,
114           maxStrategies: ctxPlan.maxStrategies,

```

```

113         planName: ctxPlan.planName,
114         isActive,
115         isBanned,
116         isCancelled,
117         needsSubscription: !isActive && !isCancelled && !isBanned
118     };
119 }
120
121 // Fallback: obtener la última suscripción y plan (no debería ocurrir si el contexto
122 está activo) latestSubscription: Subscription | null = await
123 this.subscriptionService.getUserLatestSubscription(userId);
124
125     return {
126         maxAccounts: 0,
127         maxStrategies: 0,
128         planName: 'No Plan',
129         isActive: false,
130         isBanned: false,
131         isCancelled: false,
132         needsSubscription: true
133     };
134 }
135
136 const isBanned = latestSubscription.status === UserStatus.BANNED;
137 const isCancelled = latestSubscription.status === UserStatus.CANCELLED;
138 const isActive = latestSubscription.status === UserStatus.PURCHASED ||
139     latestSubscription.status === UserStatus.CREATED ||
140     latestSubscription.status === UserStatus.PROCESSING ||
141     latestSubscription.status === UserStatus.ACTIVE;
142
143 if (isBanned || isCancelled || !isActive) {
144     return {
145         maxAccounts: 0,
146         maxStrategies: 0,
147         planName: 'Inactive Plan',
148         isActive: false,
149         isBanned,
150         isCancelled,
151         needsSubscription: true
152     };
153 }
154
155 // Get plan details from Firebase
156 const plan = await this.planService.getPlanById(latestSubscription.planId);
157
158 if (!plan) {
159     return {
160         maxAccounts: 0,
161         maxStrategies: 0,
162         planName: 'Unknown Plan',
163         isActive: false,
164         isBanned: false,
165         isCancelled: false,
166         needsSubscription: true
167     };
168 }
169
170 // Extract limitations from plan
171 const maxAccounts = plan.tradingAccounts || 1;
172 const maxStrategies = plan.strategies || 1;
173
174 return {
175     maxAccounts,
176     maxStrategies,
177     planName: plan.name,
178     isActive: true,
179     isBanned: false,
180     isCancelled: false,
181     needsSubscription: false
182 };

```

```

183     } catch (error) {
184         console.error('Error checking user limitations:', error);
185         return {
186             maxAccounts: 0,
187             maxStrategies: 0,
188             planName: 'Error',
189             isActive: false,
190             isBanned: false,
191             isCancelled: false,
192             needsSubscription: true
193         };
194     }
195 }
196
197 /**
198  * Check if user can create accounts
199  */
200 async checkAccountCreation(userId: string, currentAccountCount: number):
201 Promise<LimitationCheck> await this.checkUserLimitations(userId);
202
203 // If user needs subscription or is banned/cancelled
204 if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
205     return {
206         canCreate: false,
207         showBlockedModal: true,
208         blockedMessage: this.getBlockedMessage(limitations),
209         showUpgradeModal: false,
210         upgradeMessage: ''
211     };
212 }
213
214 // Check if user has reached account limit
215 if (currentAccountCount >= limitations.maxAccounts) {
216     return {
217         canCreate: false,
218         showUpgradeModal: true,
219         upgradeMessage: `You've reached the account limit for your ${limitations.planName}
220 plan. Move to a higher plan and keep growing your account.`,
221         blockedMessage: ''
222     };
223 }
224
225 return {
226     canCreate: true,
227     showUpgradeModal: false,
228     upgradeMessage: '',
229     showBlockedModal: false,
230     blockedMessage: ''
231 };
232 }
233
234 /**
235  * Check if user can create strategies
236  */
237 async checkStrategyCreation(userId: string, currentStrategyCount: number):
238 Promise<LimitationCheck> await this.checkUserLimitations(userId);
239
240 // If user needs subscription or is banned/cancelled
241 if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
242     return {
243         canCreate: false,
244         showBlockedModal: true,
245         blockedMessage: this.getBlockedMessage(limitations),
246         showUpgradeModal: false,
247         upgradeMessage: ''
248     };
249 }
250
251 // Check if user has reached strategy limit
252 if (currentStrategyCount >= limitations.maxStrategies) {

```

```

253     return {
254         canCreate: false,
255         showUpgradeModal: true,
256         upgradeMessage: `You've reached the strategy limit for your ${limitations.planName}
257 plan. Move to a higher plan and keep growing your account.`,
258         blockedMessage: ''
259     };
260 }
261
262     return {
263         canCreate: true,
264         showUpgradeModal: false,
265         upgradeMessage: '',
266         showBlockedModal: false,
267         blockedMessage: ''
268     };
269 }
270
271 /**
272  * Get appropriate blocked message based on user status
273  */
274 private getBlockedMessage(limitations: PlanLimitations): string {
275     if (limitations.isBanned) {
276         return 'Your account has been banned. Please contact support for assistance.';
277     }
278
279     if (limitations.isCancelled) {
280         return 'Your subscription has been cancelled. Please purchase a plan to access this
281 functionality.';
282     }
283
284     if (limitations.needsSubscription) {
285         return 'You need to purchase a plan to access this functionality.';
286     }
287
288     return 'Access denied. Please contact support for assistance.';
289 }
290
291 /**
292  * Navigate to account page for plan management
293  */
294 navigateToAccount(): void {
295     this.router.navigate(['/account']);
296 }
297
298 /**
299  * Navigate to signup page for new users
300  */
301 navigateToSignup(): void {
302     this.router.navigate(['/signup']);
303 }
304
305 /**
306  * Check if user can access a specific feature and return modal data if blocked
307  */
308 async checkFeatureAccess(
309     userId: string,
310     feature: 'accounts' | 'strategies' | 'reports',
311     currentCount?: number
312 ): Promise<{ canAccess: boolean; modalData?: ModalData }> {
313     try {
314         const limitations = await this.checkUserLimitations(userId);
315
316         // If user needs subscription or is banned/cancelled
317         if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
318             return {
319                 canAccess: false,
320                 modalData: {
321                     showModal: true,
322                     modalType: 'blocked',
323                     title: 'Access Restricted',

```

```

323         message: this.getBlockedMessage(limitations),
324         primaryButtonText: 'Go to Account',
325         onPrimaryAction: () => this.navigateToAccount()
326     }
327 };
328 }
329
330 // Check specific feature limits
331 let maxAllowed = 0;
332 let featureName = '';
333
334 switch (feature) {
335     case 'accounts':
336         maxAllowed = limitations.maxAccounts;
337         featureName = 'trading accounts';
338         break;
339     case 'strategies':
340         maxAllowed = limitations.maxStrategies;
341         featureName = 'strategies';
342         break;
343     case 'reports':
344         // Reports don't have a count limit, but need active subscription
345         return { canAccess: true };
346 }
347
348 // If user has reached the limit
349 if (currentCount !== undefined && currentCount >= maxAllowed) {
350     // Check if user is on the maximum plan (Pro plan with max limits)
351     const isProPlanWithMaxLimits = limitations.planName.toLowerCase().includes('pro') &&
352         ((feature === 'strategies' && maxAllowed === 8) ||
353         (feature === 'accounts' && maxAllowed === 10));
354
355     // If user is already on the maximum plan, don't show upgrade modal
356     if (isProPlanWithMaxLimits) {
357         return {
358             canAccess: false
359         };
360     }
361
362     // For other plans, show upgrade modal
363     return {
364         canAccess: false,
365         modalData: {
366             showModal: true,
367             modalType: 'upgrade',
368             title: 'Upgrade Required',
369             message: `You've reached the ${featureName} limit for your
370 ${limitations.planName}. Please consider moving to a higher plan and keep growing your account.`,
371             primaryButtonText: 'Upgrade',
372             secondaryButtonText: 'Cancel',
373             onPrimaryAction: () => this.navigateToAccount(),
374             onSecondaryAction: () => {}
375         }
376     };
377
378     return { canAccess: true };
379
380 } catch (error) {
381     console.error('Error checking feature access:', error);
382     return {
383         canAccess: false,
384         modalData: {
385             showModal: true,
386             modalType: 'blocked',
387             title: 'Access Restricted',
388             message: 'An error occurred while checking your access. Please try again.',
389             primaryButtonText: 'Go to Account',
390             onPrimaryAction: () => this.navigateToAccount()
391         }
392     };

```



```

393     }
394   }
395
396   /**
397    * Check if user can create accounts and return modal data if blocked
398    */
399   async checkAccountCreationWithModal(userId: string, currentAccountCount: number):
400   Promise<{ canCreate: boolean; modalData?: ModalData }> {
401     const result = await this.checkFeatureAccess(userId, 'accounts', currentAccountCount);
402     return {
403       canCreate: result.canAccess,
404       modalData: result.modalData
405     };
406   }
407
408   /**
409    * Check if user can create strategies and return modal data if blocked
410    */
411   async checkStrategyCreationWithModal(userId: string, currentStrategyCount: number):
412   Promise<{ canCreate: boolean; modalData?: ModalData }> {
413     const result = await this.checkFeatureAccess(userId, 'strategies', currentStrategyCount);
414     return {
415       canCreate: result.canAccess,
416       modalData: result.modalData
417     };
418   }
419
420   /**
421    * Check if user can access reports and return modal data if blocked
422    */
423   async checkReportAccessWithModal(userId: string): Promise<{ canAccess: boolean;
424   modalData?: ModalData }> {
425     return this.checkFeatureAccess(userId, 'reports');
426   }
427 }

```

guards\redirect-guard.guard.ts

```

1  import { CanActivateFn, Router } from '@angular/router';
2  import { AuthService } from '../features/auth/service/authService';
3  import { inject } from '@angular/core';
4  import { catchError, from, map, of, switchMap, take, tap } from 'rxjs';
5  import { Store } from '@ngrx/store';
6  import { getAuth } from 'firebase/auth';
7  import { setUserData } from '../features/auth/store/user.actions';
8
9  /**
10   * Redirect guard that handles post-authentication routing.
11   *
12   * This guard is used after successful authentication to redirect users
13   * to the appropriate page based on their role. Admins are redirected to
14   * the overview page, while regular users are redirected to the strategy page.
15   *
16   * Features:
17   * - Verifies authentication state
18   * - Checks user status (banned users are blocked)
19   * - Role-based redirection (admin !' overview, user !' strategy)
20   * - Dispatches user data to NgRx store
21   * - Shows ban alert for banned users
22   *
23   * Redirect Logic:
24   * - Admin users !' /overview
25   * - Regular users !' /strategy
26   * - Banned users !' /login (with alert)
27   * - Unauthenticated !' /login
28   *
29   * Relations:
30   * - AuthService: Checks authentication and fetches user data

```

```

31 * - Store (NgRx): Dispatches user data
32 * - Router: Handles navigation redirects
33 *
34 * @guard
35 * @function redirectGuard
36 */
37 export const redirectGuard: CanActivateFn = (route, state) => {
38   const router = inject(Router);
39   const authService = inject(AuthService);
40   const store = inject(Store);
41
42   return authService.isAuthenticated().pipe(
43     take(1),
44     switchMap((isAuth) => {
45       if (!isAuth) {
46         router.navigate(['/login']);
47         return of(false);
48       }
49
50       const user = getAuth().currentUser;
51       if (!user) {
52         router.navigate(['/login']);
53         return of(false);
54       }
55
56       return from(authService.getUserData(user.uid)).pipe(
57         tap((userData) => {
58           if (userData.status === 'banned') {
59             alert('You are banned, call support');
60             router.navigate(['/login']);
61             throw new Error('User banned');
62           }
63           store.dispatch(setUserData({ user: userData }));
64
65           // Redirección inteligente según tipo de usuario
66           if (userData.isAdmin) {
67             router.navigate(['/overview']);
68           } else {
69             router.navigate(['/strategy']);
70           }
71         })),
72         map(() => true),
73         catchError(() => {
74           router.navigate(['/login']);
75           return of(false);
76         })
77       );
78     }));
79   };
80 };
81

```