TradeSwitch App

Features TypeScript Code

Generated: 11/25/2025, 6:38:23 PM

Total Files: 72

Table of Contents

10. features\auth\service (1 files)
11. features\auth\signup (1 files)
12. features\auth\signup\components\plan-selection (1 files)
13. features\auth\store (3 files)
14. features\overview (1 files)
15. features\overview\components\top-list (1 files)
16. features\overview\components\tradeSwitch-table (1 files)
17. features\overview\models (1 files)
18. features\overview\services (1 files)
19. features\report (1 files)
20. features\report\components\calendar (1 files)
21. features\report\components\pnlGraph (1 files)
22. features\report\components\rule-short (1 files)
23. features\report\components\statCard (1 files)
24. features\report\components\trades-popup (1 files)
25. features\report\components\winLossChart (1 files)
26. features\report\models (1 files)
27. features\report\service (1 files)
28. features\report\store (3 files)
29. features\report\utils (2 files)
30. features\revenue (1 files)
31. features\revenue\components\orders-table (1 files)
32. features\revenue\components\revenue-table (1 files)
33. features\revenue\components\revenueGraph (1 files)
34. features\revenue\components\subscriptions-table (1 files)
35. features\revenue\mocks (1 files)
36. features\revenue\models (1 files)
37. features\strategy (1 files)
38. features\strategy\components\assets-allowed (1 files)
39. features\strategy\components\days-allowed (1 files)
40. features\strategy\components\hours-allowed (1 files)
41. features\strategy\components\max-daily-trades (1 files)
42. features\strategy\components\risk-per-trade (1 files)
43. features\strategy\components\risk-per-trade\models (1 files)
44. features\strategy\components\risk-reward (1 files)
45. features\strategy\edit-strategy (1 files)
46. features\strategy\edit-strategy\components\edit-assets-allowed (1 files)
47. features\strategy\edit-strategy\components\edit-days-allowed (1 files)
48. features\strategy\edit-strategy\components\edit-hours-allowed (1 files)
49. features\strategy\edit-strategy\components\edit-max-daily-trades (1 files)
50. features\strategy\edit-strategy\components\edit-risk-per-trade (1 files)
51. features\strategy\edit-strategy\components\edit-risk-per-trade\models (1 files)
52. features\strategy\edit-strategy\components\edit-risk-reward (1 files)
53. features\strategy\models (1 files)
54. features\strategy\service (1 files)
55. features\strategy\services (2 files)
56. features\strategy\store (3 files)

57. features\trading-accounts (1 files)
58. features\trading-accounts\components\accounts-table (1 files)
59. features\trading-accounts\components\show-confirmation (1 files)
60. features\users-details (1 files)
61. features\users-details\components\create-user-role-popup (1 files)
62. features\users-details\components\user-modal (1 files)
63. features\users-details\components\users-table (1 files)

Ø=ÜÁ features\account

Ø=ÜÄ features\account\account.component.ts

```typescript
import { Component, OnInit } from '@angular/core';
import { User } from '../overview/models/overview';
import { Store } from '@ngrx/store';
import { SettingsService } from '../strategy/service/strategy.service';
import { ReportService } from '../report/service/report.service';
import { CommonModule } from '@angular/common';
import { PlanSettingsComponent } from './components/plan-management/plan-settings.component';
import { ProfileDetailsComponent } from './components/profile-details/profile-details.component';
import { SubscriptionHistoryComponent } from './components/subscription-history/subscription-history.component';
import { PlanDetails } from './models/account-settings';
import { MOCK_PLAN_DETAILS } from './mocks/account-mocks';
import { AppContextService } from '../../shared/context';
import { ActivatedRoute } from '@angular/router';

/**
 * Main component of the user account module.
 *
 * This component acts as the main container that manages navigation between
 * different sections of the user account:
 * - Profile Details: Profile details and configuration
 * - Plan Management: Plan and subscription management
 *
 * Related to:
 * - ProfileDetailsComponent: Displays and allows editing of profile data
 * - PlanSettingsComponent: Manages user subscription plans
 * - AppContextService: Gets current user data
 * - ActivatedRoute: Reads URL parameters to select the correct tab
 *
 * @component
 * @selector app-account
 * @standalone true
 */
@Component({
  selector: 'app-account',
  imports: [CommonModule, PlanSettingsComponent, ProfileDetailsComponent],
  templateUrl: './account.component.html',
  styleUrl: './account.component.scss',
  standalone: true,
})
export class AccountComponent implements OnInit {
  /** Current user in the system */
  user: User | null = null;

  /** Index of the selected tab (0: Profile Details, 1: Plan Management) */
  selectedIndex: number = 0;

  /** Array of available tabs in the interface */
  tabs: { label: string }[] = [
    { label: 'Profile Details' },
    { label: 'Plan Management' },
    /*{ label: 'Subscription History' },*/
  ];

  /** Selected plan details (currently uses mock data) */
  selectedPlanDetails: PlanDetails | null = null;

  /**
   * Constructor for AccountComponent
   *
   * @param store - NgRx Store to access global state
   * @param strategySvc - Strategy service (injected but not currently used)
   * @param reportSvc - Report service (injected but not currently used)
   * @param appContext - Application context service to get user data
   * @param route - ActivatedRoute to read URL parameters
```

```
   */
  constructor(
    private store: Store,
    private strategySvc: SettingsService,
    private reportSvc: ReportService,
    private appContext: AppContextService,
    private route: ActivatedRoute
  ) {}

  /**
   * Initializes the component on load.
   *
   * Performs the following actions:
   * 1. Loads plan details from mocks
   * 2. Checks URL query parameters to select the correct tab
   *    (if ?tab=plan exists, selects the Plan Management tab)
   * 3. Subscribes to user changes from the application context
   * 4. If no user is available, attempts to load it
   *
   * @memberof AccountComponent
   */
  ngOnInit(): void {
    this.selectedPlanDetails = MOCK_PLAN_DETAILS;

    // Verificar query parameters para seleccionar la pestaña correcta
    this.route.queryParams.subscribe(params => {
      if (params['tab'] === 'plan') {
        this.selectedIndex = 1; // Plan Management tab
      }
    });

    // Suscribirse a los datos del usuario desde el contexto
    this.appContext.currentUser$.subscribe(user => {
      this.user = user;
    });

    // Cargar datos del usuario si no están disponibles
    if (!this.user) {
      this.appContext.setLoading('user', true);
      // Aquí se podría implementar la carga de datos del usuario si es necesario
      this.appContext.setLoading('user', false);
    }
  }

  /**
   * Changes the selected tab in the interface.
   *
   * This method is executed when the user clicks on a different tab.
   * Updates the selected index, which causes the corresponding component
   * to be displayed in the template.
   *
   * @param index - Index of the tab to select (0: Profile Details, 1: Plan Management)
   * @memberof AccountComponent
   */
  selectTypeData(index: number): void {
    this.selectedIndex = index;
  }
}

```

Ø=ÜÁ features\account\components\plan-management

Ø=ÜÄ features\account\components\plan-management\plan-settings.component.ts

```
import { Component, Input, OnInit, inject } from '@angular/core';

import { CommonModule } from '@angular/common';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../../strategy/service/strategy.service';
import { ReportService } from '../../../report/service/report.service';
import { User } from '../../../overview/models/overview';
import { PlanCard, PlanDetails } from '../../models/account-settings';
import { PLANS } from '../../mocks/account-mocks';
import { Subscription, SubscriptionService } from '../../../../shared/services/subscription-service';
import { PlanService } from '../../../../shared/services/planService';
import { AuthService } from '../../../auth/service/authService';
import { Plan } from '../../../../shared/services/planService';
import { selectUser } from '../../../auth/store/user.selectios';
import { SubscriptionProcessingComponent } from '../../../../shared/components/subscription-processing/subscription-processing.component';
import { OrderSummaryComponent } from '../../../../shared/components/order-summary/order-summary.component';
import { UserStatus } from '../../../overview/models/overview';
import { AppContextService } from '../../../../shared/context/context';
import { LoadingSpinnerComponent } from '../../../../shared/components/loading-spinner/loading-spinner.component';
import { StripeLoaderPopupComponent } from '../../../../shared/pop-ups/stripe-loader-popup/stripe-loader-popup.component';
/**
 * Component for managing user subscription plans.
 *
 * This component allows the user to:
 * - View their current plan and renewal details
 * - Compare and switch between different available plans
 * - Validate downgrades before switching to a lower plan
 * - Manage their subscription through the Stripe portal
 * - Cancel their current plan
 *
 * Related to:
 * - AccountComponent: Receives planDetails as Input
 * - SubscriptionService: Gets and updates user subscriptions
 * - PlanService: Gets information about available plans
 * - AuthService: Gets authentication tokens for API calls
 * - AppContextService: Accesses global plans and user data
 * - Stripe: Integration for checkout and subscription management portal
 *
 * Main flow:
 * 1. On initialization, loads user plan and available plans
 * 2. Builds plan cards from global context data
 * 3. Calculates renewal dates and remaining days
 * 4. Handles plan changes with downgrade validation
 * 5. Integrates with Stripe for payments and subscription management
 *
 * @component
 * @selector app-plan-settings
 * @standalone true
 */
@Component({
  selector: 'app-plan-settings',
  imports: [CommonModule, LoadingSpinnerComponent, StripeLoaderPopupComponent /*SubscriptionProcessingComponent,OrderSummaryComponent*/],
  templateUrl: './plan-settings.component.html',
  styleUrl: './plan-settings.component.scss',
  standalone: true,
})
export class PlanSettingsComponent implements OnInit {
  /** Plan details received from parent component (AccountComponent) */
  @Input() planDetails: PlanDetails | null = null;

  /** Array of available plan cards to display in the interface */
  plansData: PlanCard[] = [];

```

```
  /** Current user plan obtained from the service */
  userPlan: Plan | undefined = undefined;

  /** Plan renewal date formatted as string */
  renewalDate: string = '';

  /** Remaining days until next renewal */
  remainingDays: number = 0;

  /** Flag to determine if user has free plan (shows N/A for renewal) */
  isFreePlan: boolean = false;

  // Estado de carga inicial
  initialLoading: boolean = true;


  user: User | null = null;
  selectedIndex: number = 0;
  tabs: { label: string }[] = [
    { label: 'Profile Details' },
    { label: 'Plan Management' },
    { label: 'Billing Management' },
  ];

  // Estados para cancelar plan
  showCancelPlanProcessing = false;

  // Estados para validación de downgrade
  showDowngradeValidation = false;
  downgradeValidationData: {
    targetPlan: string;
    currentAccounts: number;
    maxAccounts: number;
    currentStrategies: number;
    maxStrategies: number;
    accountsToDelete: number;
    strategiesToDelete: number;
  } | null = null;

  // Estados para pop-up de carga y error de redirección
  showRedirectLoading = false;
  showRedirectError = false;
  redirectErrorMessage = '';
  private windowCheckInterval: any = null;

  // Inyectar servicios
  private subscriptionService = inject(SubscriptionService);
  private planService = inject(PlanService);
  private authService = inject(AuthService);
  private appContext = inject(AppContextService);

  constructor(
    private store: Store,
    private strategySvc: SettingsService,
    private reportSvc: ReportService
  ) {}

  /**
   * Initializes the component on load.
   *
   * Performs the following actions in order:
   * 1. Subscribes to changes in global plans from context
   * 2. Attempts to build plan cards immediately
   * 3. If no plans are loaded, loads them manually from PlanService
   * 4. Loads current user plan from SubscriptionService
   *
   * @async
   * @memberof PlanSettingsComponent
   */
  async ngOnInit(): Promise<void> {
```

```
    this.initialLoading = true;

    try {
      // Suscribirse a cambios en los planes globales
      this.appContext.subscribeToGlobalPlansChanges().subscribe(plans => {
        if (plans.length > 0) {
          this.buildPlansData();
        }
      });

      // También intentar construir inmediatamente por si ya están cargados
      this.buildPlansData();

      // Si no hay planes, intentar cargarlos manualmente
      if (this.appContext.globalPlans().length === 0) {
        await this.loadPlansManually();
      }

      await this.loadUserPlan();
    } finally {
      this.initialLoading = false;
    }
  }

  /**
   * Loads the current user plan from subscription.
   *
   * This method:
   * 1. Gets the current user from the store
   * 2. Finds the user's most recent subscription
   * 3. Gets the plan associated with the subscription from PlanService
   * 4. Calculates renewal date and remaining days
   * 5. If there's no subscription or plan, sets default free plan
   *
   * Related to:
   * - SubscriptionService.getUserLatestSubscription(): Gets user subscription
   * - PlanService.getPlanById(): Gets plan details by ID
   * - calculateRenewalDate(): Calculates renewal date
   * - setDefaultFreePlan(): Sets free plan if there's no subscription
   *
   * @private
   * @async
   * @memberof PlanSettingsComponent
   */
  private async loadUserPlan(): Promise<void> {
    try {
      // Obtener el usuario actual
      this.getUserData();
      if (!this.user) {
        console.error(''L User not found');
        return;
      }

      // Obtener la suscripción del usuario
      const subscription = await
this.subscriptionService.getUserLatestSubscription(this.user.id);
      if (subscription && subscription.planId) {
        // Buscar el plan por ID
        const plan: Plan | undefined = await
this.planService.getPlanById(subscription.planId);
        if (plan) {
          this.userPlan = plan;
          this.isFreePlan = plan.name.toLowerCase() === 'free';

          // Usar periodEnd si existe, sino usar created_at
          if (subscription.periodEnd) {
            this.calculateRenewalDate(subscription.periodEnd);
          } else {
            this.calculateRenewalDate(subscription.created_at);
          }
        } else {
```

```
          // Si no se encuentra el plan, usar plan gratuito por defecto
          this.setDefaultFreePlan();
        }
      } else {
        // Si no hay suscripción, usar plan gratuito por defecto
        this.setDefaultFreePlan();
      }
    } catch (error) {
      console.error('Error loading user plan:', error);
      this.setDefaultFreePlan();
    }
  }


  /**
   * Builds the array of plan cards from global context plans.
   *
   * This method transforms plans obtained from AppContextService into
   * PlanCard objects used to display cards in the interface.
   *
   * For each plan:
   * - Assigns price and period
   * - Marks the second plan as "most popular"
   * - Assigns icons and colors based on position
   * - Builds the features array (trading accounts, strategies, etc.)
   * - Defines the CTA button text
   *
   * Related to:
   * - AppContextService.orderedPlans(): Gets ordered plans
   * - AppContextService.getPlanLimits(): Gets limits for each plan
   *
   * @private
   * @memberof PlanSettingsComponent
   */
  private buildPlansData(): void {
    // Usar planes del contexto global
    const orderedPlans = this.appContext.orderedPlans();

    if (orderedPlans.length === 0) {
      return;
    }

    // Construir plansData desde los planes ordenados del contexto
    this.plansData = orderedPlans.map((plan, index) => ({
      name: plan.name,
      price: parseInt(plan.price) || 0,
      period: '/month',
      mostPopular: index === 1, // Marcar el segundo plan como más popular (Starter)
      icon: index === 0 ? 'triangle' : index === 1 ? 'circle' : 'square',
      color: index === 0 ? '#4b7ee8' : index === 1 ? '#4b7ee8' : '#d1ff81',
      features: [
        { label: 'Trading Accounts', value: plan.tradingAccounts.toString() },
        { label: 'Strategies', value: plan.strategies.toString() },
        { label: 'Consistency Rules', value: 'YES' },
        { label: 'Trading Journal', value: 'YES' },
        { label: 'Live Statistics', value: 'YES' }
      ],
      cta: `Get ${plan.name} Now`
    }));
  }

  /**
   * Loads plans manually from PlanService if they're not in context.
   *
   * This method runs as a fallback when plans are not available
   * in the global context. Loads all plans from PlanService and
   * sets them in the context for future use.
   *
   * Related to:
   * - PlanService.getAllPlans(): Gets all available plans
```

```
   * - AppContextService.setGlobalPlans(): Sets plans in context
   * - buildPlansData(): Builds cards after loading
   *
   * @private
   * @async
   * @memberof PlanSettingsComponent
   */
  private async loadPlansManually(): Promise<void> {
    try {
      const plans = await this.planService.getAllPlans();
      this.appContext.setGlobalPlans(plans);
      this.buildPlansData();
    } catch (error) {
      console.error(''L Error cargando planes manualmente:', error);
    }
  }

  /**
   * Gets current user data from NgRx store.
   *
   * Subscribes to selectUser selector to get current user
   * and update the component's user property.
   *
   * Related to:
   * - Store.select(selectUser): NgRx selector to get user
   *
   * @private
   * @memberof PlanSettingsComponent
   */
  private getUserData() {
    this.store.select(selectUser).subscribe({
      next: (user) => {
        this.user = user.user;
      },
      error: (err) => {
        console.error('Error fetching user data', err);
      },
    });
  }

  /**
   * Sets the free plan as the user's default plan.
   *
   * This method runs when:
   * - User doesn't have an active subscription
   * - Plan associated with subscription cannot be found
   * - An error occurs loading user plan
   *
   * Searches for "Free" plan in global context, or creates a default one
   * if not available. Sets renewal date as "N/A"
   * and remaining days to 0.
   *
   * Related to:
   * - AppContextService.getPlanByName(): Searches for Free plan in context
   *
   * @private
   * @memberof PlanSettingsComponent
   */
  private setDefaultFreePlan(): void {
    // Buscar el plan Free en los planes del contexto global
    const freePlan = this.appContext.getPlanByName('Free');

    if (freePlan) {
      this.userPlan = freePlan;
    } else {
      // Fallback si no se encuentra el plan Free en el contexto
      this.userPlan = {
        id: 'free',
        name: 'Free',
        price: '0',
```

```
        strategies: 1,
        tradingAccounts: 1,
        createdAt: new Date(),
        updatedAt: new Date()
      };
    }
    this.isFreePlan = true;
    this.renewalDate = 'N/A';
    this.remainingDays = 0;
  }

  /**
   * Calculates renewal date and remaining days until renewal.
   *
   * This method:
   * 1. Checks if user has free plan (returns N/A if so)
   * 2. Converts periodEnd to Date object (handles both Firebase Timestamp and Date)
   * 3. Formats renewal date in readable format
   * 4. Calculates remaining days from today until renewal date
   * 5. If date has passed, sets days to 0
   *
   * Related to:
   * - loadUserPlan(): Called after getting user subscription
   *
   * @private
   * @param periodEnd - Subscription period end date (can be Firebase Timestamp or Date)
   * @memberof PlanSettingsComponent
   */
  private calculateRenewalDate(periodEnd?: any): void {
    // Si es plan Free, no calcular nada
    if (this.isFreePlan) {
      this.renewalDate = 'N/A';
      this.remainingDays = 0;
      return;
    }

    let renewalDate: Date;

    if (periodEnd) {
      // Usar periodEnd de la subscription
      renewalDate = periodEnd.toDate ? periodEnd.toDate() : new Date(periodEnd);
    } else {
      // Fallback: usar fecha actual
      renewalDate = new Date();
    }

    // Formatear fecha de renovación
    this.renewalDate = renewalDate.toLocaleDateString('en-US', {
      year: 'numeric',
      month: 'long',
      day: 'numeric'
    });

    // Calcular días restantes desde hoy hasta la fecha de renovación
    const today = new Date();
    today.setHours(0, 0, 0, 0); // Reset horas para comparación precisa
    renewalDate.setHours(0, 0, 0, 0);

    const timeDiff = renewalDate.getTime() - today.getTime();
    this.remainingDays = Math.ceil(timeDiff / (1000 * 3600 * 24));

    // Si ya pasó la fecha de renovación, mostrar 0 días
    if (this.remainingDays < 0) {
      this.remainingDays = 0;
    }
  }

  selectTypeData(index: number): void {
    this.selectedIndex = index;
  }
```

```
415
416     /**
417      * Gets capitalized current plan name.
418      *
419      * Helper to display plan name in readable format
420      * (first letter uppercase, rest lowercase).
421      *
422      * @returns Capitalized plan name or "Free Plan" if no plan
423      * @memberof PlanSettingsComponent
424      */
425     getCapitalizedPlanName(): string {
426       if (!this.userPlan?.name) return 'Free Plan';
427       const name = this.userPlan.name;
428       return name.charAt(0).toUpperCase() + name.slice(1).toLowerCase();
429     }
430
431     /**
432      * Capitalizes any plan name.
433      *
434      * Generic helper to format plan names
435      * (first letter uppercase, rest lowercase).
436      *
437      * @param planName - Plan name to capitalize
438      * @returns Capitalized plan name or empty string if not provided
439      * @memberof PlanSettingsComponent
440      */
441     capitalizePlanName(planName: string): string {
442       if (!planName) return '';
443       return planName.charAt(0).toUpperCase() + planName.slice(1).toLowerCase();
444     }
445
446     /**
447      * Checks if a plan is the user's current plan.
448      *
449      * Compares the provided plan name with the user's current
450      * plan name (case-insensitive comparison).
451      *
452      * @param planName - Name of plan to check
453      * @returns true if plan is current plan, false otherwise
454      * @memberof PlanSettingsComponent
455      */
456     isCurrentPlan(planName: string): boolean {
457       if (!this.userPlan) return false;
458       return this.userPlan.name.toLowerCase() === planName.toLowerCase();
459     }
460
461     /**
462      * Gets the number of trading accounts allowed for a plan.
463      *
464      * Searches for plan limits in global context and returns
465      * the number of allowed trading accounts.
466      *
467      * Related to:
468      * - AppContextService.getPlanLimits(): Gets plan limits
469      *
470      * @param planName - Plan name
471      * @returns Number of trading accounts as string (default "1")
472      * @memberof PlanSettingsComponent
473      */
474     getTradingAccounts(planName: string): string {
475       // Usar datos del contexto global
476       const limits = this.appContext.getPlanLimits(planName);
477       return limits ? limits.tradingAccounts.toString() : '1';
478     }
479
480     /**
481      * Gets the number of strategies allowed for a plan.
482      *
483      * Searches for plan limits in global context and returns
484      * the number of allowed strategies.
```

```
   *
   * Related to:
   * - AppContextService.getPlanLimits(): Gets plan limits
   *
   * @param planName - Plan name
   * @returns Number of strategies as string (default "1")
   * @memberof PlanSettingsComponent
   */
  getStrategies(planName: string): string {
    // Usar datos del contexto global
    const limits = this.appContext.getPlanLimits(planName);
    return limits ? limits.strategies.toString() : '1';
  }

  /**
   * Gets button text based on plan status.
   *
   * Returns:
   * - "Current plan" if plan is user's current plan
   * - "Change plan" for all other cases
   *
   * @param planName - Plan name
   * @returns Corresponding button text
   * @memberof PlanSettingsComponent
   */
  getButtonText(planName: string): string {
    // Verificar si el plan de la card es el plan actual del usuario
    const currentPlanName = this.userPlan?.name.toLowerCase();
    const cardPlanName = planName.toLowerCase();

    // Si el plan de la card coincide con el plan actual del usuario
    if (currentPlanName === cardPlanName) {
      return 'Current plan';
    }

    // Para todos los demás casos
    return 'Change plan';
  }

  /**
   * Determines if a plan's button should be disabled.
   *
   * Button is disabled if:
   * - Plan is user's current plan
   *
   * @param planName - Plan name
   * @returns true if button should be disabled, false otherwise
   * @memberof PlanSettingsComponent
   */
  isButtonDisabled(planName: string): boolean {
    const isCurrentPlanFree = this.userPlan?.name.toLowerCase() === 'free';

    // Solo deshabilitar el botón FREE cuando el usuario tiene plan FREE
    if (this.userPlan?.name.toLowerCase() === planName.toLowerCase()) {
      return true;
    }

    return false;
  }

  /**
   * Handles plan change when user selects a new plan.
   *
   * This method is the main entry point for changing plans.
   * Performs the following actions:
   * 1. Checks if selected plan is current plan (does nothing if so)
   * 2. Validates if it's a downgrade and checks if user has resources exceeding target plan
   * 3. If downgrade and there are excess resources, shows validation modal
   * 4. If current plan is FREE and target is also FREE, does nothing
   * 5. If current plan is FREE, creates Stripe checkout session
```

```
   * 6. If current plan is NOT FREE, opens Stripe portal for management
   *
   * Related to:
   * - isCurrentPlan(): Checks if it's current plan
   * - isDowngrade(): Determines if it's a downgrade
   * - validateDowngrade(): Validates if downgrade is possible
   * - createCheckoutSession(): Creates checkout session for paid plans
   * - openStripePortal(): Opens Stripe portal for subscription management
   *
   * @async
   * @param plan - Selected plan card
   * @memberof PlanSettingsComponent
   */
  async onPlanChange(plan: PlanCard): Promise<void> {
    try {
      // Verificar si es el plan actual
      if (this.isCurrentPlan(plan.name)) {
        return; // No hacer nada si es el plan actual
      }

      // Validar si es un downgrade y si el usuario tiene recursos que exceden el plan de destino
      const isDowngrade = this.isDowngrade(plan.name);

      if (isDowngrade) {
        const validationResult = await this.validateDowngrade(plan.name);

        if (!validationResult.canDowngrade) {
          this.showDowngradeValidationModal(validationResult);
          return;
        }
      }

      // Verificar si el plan actual es FREE
      const isCurrentPlanFree = this.userPlan?.name.toLowerCase() === 'free';
      const isTargetPlanFree = plan.name.toLowerCase() === 'free';

      // Si el plan actual es FREE y el plan de destino también es FREE, no hacer nada
      if (isCurrentPlanFree && isTargetPlanFree) {
        return; // No hacer nada si ambos son Free
      }

      // Si llegamos aquí, significa que puede hacer el cambio de plan
      // Mostrar pop-up de carga solo para planes de pago
      this.showRedirectLoading = true;

      // Variable para controlar si hay error
      let hasError = false;
      let errorMessage = '';

      try {
        if (isCurrentPlanFree) {
          // Si el plan actual es FREE y hace click en otro plan !' crear checkout session
          await this.createCheckoutSession(plan.name);
        } else {
          // Si el plan actual NO es FREE !' abrir portal de Stripe
          await this.openStripePortal();
        }
      } catch (error) {
        // Marcar que hay error pero no mostrar pop-up aún
        hasError = true;
        errorMessage = 'Error redirecting to payment. Please try again.';
        console.error('Error during plan change:', error);
      }

      // Esperar mínimo 2 segundos antes de mostrar error o ocultar loader
      setTimeout(() => {
        if (hasError) {
          // Si hay error, mostrar pop-up de error
          this.showRedirectLoading = false;
          this.showRedirectError = true;
```

```
          this.redirectErrorMessage = errorMessage;

          // Limpiar intervalo si existe
          if (this.windowCheckInterval) {
            clearInterval(this.windowCheckInterval);
            this.windowCheckInterval = null;
          }
        }
        // Si no hay error, el loader se ocultará automáticamente cuando se cierre la ventana
      }, 2000);
    } catch (error) {
      console.error('Error processing plan change:', error);
      // Eliminar el alert y manejar el error de forma más elegante
      console.error('Error processing your request. Please try again.');
    }
  }

  /**
   * Creates a Stripe checkout session for a new plan.
   *
   * This method runs when user has FREE plan and wants to
   * switch to a paid plan. Performs:
   * 1. Gets complete plan from context to obtain planPriceId
   * 2. Gets Firebase authentication token
   * 3. Makes POST request to API to create checkout session
   * 4. Redirects user to Stripe checkout URL
   *
   * Related to:
   * - AppContextService.getPlanByName(): Gets plan by name
   * - AuthService.getBearerTokenFirebase(): Gets authentication token
   * - API: https://api.tradeswitch.io/payments/create-checkout-session
   *
   * @private
   * @async
   * @param planName - Selected plan name
   * @throws Error if planPriceId is not found or session creation fails
   * @memberof PlanSettingsComponent
   */
  private async createCheckoutSession(planName: string): Promise<void> {
    try {
      // Obtener el plan completo del contexto para obtener el priceId
      const selectedPlan = this.appContext.getPlanByName(planName);

      if (!selectedPlan || !selectedPlan.planPriceId) {
        throw new Error('Plan price ID not found');
      }

      // Obtener el token de Firebase
      const bearerTokenFirebase = await
this.authService.getBearerTokenFirebase(this.user?.id || '');
      // Crear checkout session
      const response = await fetch('https://api.tradeswitch.io/payments/create-checkout-
session', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Bearer ${bearerTokenFirebase}`
        },
        body: JSON.stringify({
          priceId: selectedPlan.planPriceId,
        })
      });

      if (!response.ok) {
        const errorText = await response.text();
        throw new Error(`Error creating checkout session: ${response.status}
${response.statusText} - ${errorText}`);
      }

      const responseData = await response.json();
      const checkoutUrl = responseData.body?.url || responseData.url;

```

```
695       if (!checkoutUrl) {
696         throw new Error('Checkout URL not found in response');
697       }
698
699       // Redirigir a la página de checkout
700       window.location.href = checkoutUrl;
701
702     } catch (error) {
703       console.error('Error creating checkout session:', error);
704       // No ocultar el loader aquí, dejar que el timeout de 2 segundos lo maneje
705       throw error;
706     }
707   }
708
709   /**
710    * Opens Stripe subscription management portal in a new window.
711    *
712    * This method runs when user has a paid plan and wants to
713    * manage their subscription. Performs:
714    * 1. Gets Firebase authentication token
715    * 2. Makes POST request to API to create portal session
716    * 3. Opens portal in a new window
717    * 4. Monitors if window closes to hide loader
718    * 5. Has a safety timeout of 8 seconds
719    *
720    * Related to:
721    * - AuthService.getBearerTokenFirebase(): Gets authentication token
722    * - API: https://api.tradeswitch.io/payments/create-portal-session
723    * - windowCheckInterval: Interval to check if window closed
724    *
725    * @private
726    * @async
727    * @throws Error if session creation fails or window cannot be opened
728    * @memberof PlanSettingsComponent
729    */
730   private async openStripePortal(): Promise<void> {
731     try {
732       const bearerTokenFirebase = await
733 this.authService.getBearerTokenFirebase(this.user?.id || '');
734       const response = await fetch('https://api.tradeswitch.io/payments/create-portal-
735 session', {
736         method: 'POST',
737         headers: {
738           'Content-Type': 'application/json',
739           'Authorization': `Bearer ${bearerTokenFirebase}`
740         },
741         body: JSON.stringify({
742           userId: this.user?.id
743         })
744       });
745
746       if (!response.ok) {
747         throw new Error('Error creating portal session');
748       }
749
750       const responseData = await response.json();
751       const portalSessionUrl = responseData.body?.url || responseData.url;
752
753       if (!portalSessionUrl) {
754         throw new Error('Portal session URL not found in response');
755       }
756
757       // Abrir portal en nueva ventana
758       const newWindow = window.open(portalSessionUrl, '_blank');
759
760       // Verificar si la ventana se abrió correctamente
761       if (!newWindow || newWindow.closed || typeof newWindow.closed == 'undefined') {
762         throw new Error('Failed to open Stripe portal. Please check your pop-up blocker.');
763       }
764
       // Verificar periódicamente si la ventana sigue abierta
```

```
      this.windowCheckInterval = setInterval(() => {
        if (newWindow.closed) {
          // La ventana se cerró, ocultar loading
          clearInterval(this.windowCheckInterval);
          this.showRedirectLoading = false;
        }
      }, 500);

      // Timeout de seguridad: si después de 10 segundos no se ha cerrado la ventana, ocultar loading
      setTimeout(() => {
        if (this.windowCheckInterval) {
          clearInterval(this.windowCheckInterval);
        }
        this.showRedirectLoading = false;
      }, 8000);

    } catch (error) {
      console.error('Error opening Stripe portal:', error);
      // No ocultar el loader aquí, dejar que el timeout de 2 segundos lo maneje
      throw error;
    }
  }

  /**
   * Shows processing modal to cancel plan.
   *
   * This method runs when user clicks the cancel plan button.
   * Shows a confirmation modal.
   *
   * @memberof PlanSettingsComponent
   */
  onCancelPlan(): void {
    this.showCancelPlanProcessing = true;
  }

  /**
   * Confirms and executes user plan cancellation.
   *
   * This method:
   * 1. Gets user's most recent subscription
   * 2. Updates subscription with CANCELLED status and empty planId
   * 3. Reloads user plan data
   *
   * Related to:
   * - SubscriptionService.getUserLatestSubscription(): Gets subscription
   * - SubscriptionService.updateSubscription(): Updates subscription
   * - loadUserPlan(): Reloads plan after cancellation
   *
   * @async
   * @memberof PlanSettingsComponent
   */
  async confirmCancelPlan(): Promise<void> {
    if (!this.user) {
      return;
    }

    try {
      // Obtener la suscripción actual del usuario
      const subscriptions = await this.subscriptionService.getUserLatestSubscription(this.user.id);
      if (subscriptions) {
        // Obtener la suscripción más reciente
        const latestSubscription = subscriptions;

        // Actualizar la suscripción con status CANCELLED y planId vacío
        await this.subscriptionService.updateSubscription(this.user.id, latestSubscription.id, {
          status: userStatus.CANCELLED,
          planId: ''
        });

```

```
        // Recargar los datos del usuario
        await this.loadUserPlan();

      } else {
        console.error('No active subscription found to cancel.');
      }

    } catch (error) {
      console.error('Error cancelling plan. Please try again.');
    } finally {
      this.showCancelPlanProcessing = false;
    }
  }

  /**
   * Cancels the plan cancellation process.
   *
   * Hides processing modal without performing any action.
   *
   * @memberof PlanSettingsComponent
   */
  cancelCancelPlan(): void {
    this.showCancelPlanProcessing = false;
  }

  /**
   * Opens Stripe portal to manage subscription.
   *
   * Similar to openStripePortal(), but runs from a specific
   * "Manage Subscription" button. Opens portal in a new window.
   *
   * Related to:
   * - AuthService.getBearerTokenFirebase(): Gets authentication token
   * - API: https://api.tradeswitch.io/payments/create-portal-session
   *
   * @async
   * @memberof PlanSettingsComponent
   */
  async onManageSubscription(): Promise<void> {
    try {
      const bearerTokenFirebase = await
this.authService.getBearerTokenFirebase(this.user?.id || '');
      const response = await fetch('https://api.tradeswitch.io/payments/create-portal-
session', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Bearer ${bearerTokenFirebase}`
        },
        body: JSON.stringify({
          userId: this.user?.id
        })
      });

      if (!response.ok) {
        throw new Error('Error creating portal session');
      }

      const responseData = await response.json();
      const portalSessionUrl = responseData.body?.url || responseData.url;

      if (!portalSessionUrl) {
        throw new Error('Portal session URL not found in response');
      }

      window.open(portalSessionUrl, '_blank');
    } catch (error) {
      console.error('Error opening Stripe portal:', error);
      // Manejar el error de forma más elegante sin alert
    }
  }
```

```
905
906     /**
907      * Determines if a plan change is a downgrade (change to a lower plan).
908      *
909      * Compares current plan level with target plan level.
910      * Levels are: Free (1), Starter (2), Pro (3).
911      *
912      * Related to:
913      * - getPlanLevel(): Gets numeric level of a plan
914      *
915      * @private
916      * @param targetPlanName - Target plan name
917      * @returns true if it's a downgrade, false otherwise
918      * @memberof PlanSettingsComponent
919      */
920     private isDowngrade(targetPlanName: string): boolean {
921       if (!this.userPlan) return false;
922
923       const currentPlanLevel = this.getPlanLevel(this.userPlan.name);
924       const targetPlanLevel = this.getPlanLevel(targetPlanName);
925
926       return targetPlanLevel < currentPlanLevel;
927     }
928
929     /**
930      * Gets numeric level of a plan for comparison.
931      *
932      * Levels are:
933      * - Free: 1
934      * - Starter: 2
935      * - Pro: 3
936      *
937      * @private
938      * @param planName - Plan name
939      * @returns Numeric level of plan (default 1)
940      * @memberof PlanSettingsComponent
941      */
942     private getPlanLevel(planName: string): number {
943       const planLevels: { [key: string]: number } = {
944         'free': 1,
945         'starter': 2,
946         'pro': 3
947       };
948       return planLevels[planName.toLowerCase()] || 1;
949     }
950
951     /**
952      * Validates if user can downgrade to a specific plan.
953      *
954      * This method checks if user has resources (trading accounts
955      * or strategies) that exceed target plan limits. If so,
956      * downgrade is not allowed until user removes excess resources.
957      *
958      * Performs:
959      * 1. Gets target plan limits
960      * 2. Loads current user data (accounts and strategies)
961      * 3. Calculates how many resources must be deleted
962      * 4. Determines if downgrade is possible
963      *
964      * Related to:
965      * - getTradingAccounts(): Gets account limit of target plan
966      * - getStrategies(): Gets strategy limit of target plan
967      * - AuthService.getUserDataForValidation(): Gets current user data
968      *
969      * @private
970      * @async
971      * @param targetPlanName - Target plan name
972      * @returns Object with validation information (canDowngrade, resources to delete, etc.)
973      * @memberof PlanSettingsComponent
974      */
```

```
  private async validateDowngrade(targetPlanName: string): Promise<{
    canDowngrade: boolean;
    targetPlan: string;
    currentAccounts: number;
    maxAccounts: number;
    currentStrategies: number;
    maxStrategies: number;
    accountsToDelete: number;
    strategiesToDelete: number;
  }> {
    if (!this.user?.id) {
      throw new Error('User ID not available');
    }

    // Obtener límites del plan de destino usando la lógica existente
    const targetMaxAccountsStr = this.getTradingAccounts(targetPlanName);
    const targetMaxStrategiesStr = this.getStrategies(targetPlanName);

    const targetMaxAccounts = parseInt(targetMaxAccountsStr);
    const targetMaxStrategies = parseInt(targetMaxStrategiesStr);

    // Cargar datos actuales del usuario directamente desde Firebase
    const userData = await this.authService.getUserDataForValidation(this.user.id);
    const currentAccounts = userData.accounts.length;
    const currentStrategies = userData.strategies.length;

    const accountsToDelete = Math.max(0, currentAccounts - targetMaxAccounts);
    const strategiesToDelete = Math.max(0, currentStrategies - targetMaxStrategies);

    const canDowngrade = accountsToDelete === 0 && strategiesToDelete === 0;

    return {
      canDowngrade,
      targetPlan: targetPlanName,
      currentAccounts,
      maxAccounts: targetMaxAccounts,
      currentStrategies,
      maxStrategies: targetMaxStrategies,
      accountsToDelete,
      strategiesToDelete
    };
  }

  /**
   * Shows downgrade validation modal.
   *
   * Sets validation data and shows modal that informs
   * user about resources they must delete before downgrading.
   *
   * @private
   * @param validationData - Validation data obtained from validateDowngrade()
   * @memberof PlanSettingsComponent
   */
  private showDowngradeValidationModal(validationData: any): void {
    this.downgradeValidationData = validationData;
    this.showDowngradeValidation = true;
  }

  /**
   * Closes downgrade validation modal.
   *
   * Hides modal and clears validation data.
   *
   * @memberof PlanSettingsComponent
   */
  closeDowngradeValidation(): void {
    this.showDowngradeValidation = false;
    this.downgradeValidationData = null;
  }

```

```
  /**
   * Navigates to resource management pages.
   *
   * This method runs when user wants to delete resources
   * before downgrading. Currently only shows a console message.
   *
   * TODO: Implement real navigation to resource management pages.
   *
   * @memberof PlanSettingsComponent
   */
  goToManageResources(): void {
    this.showDowngradeValidation = false;
    this.downgradeValidationData = null;

    // Navegar a las páginas de gestión de recursos
    // TODO: Implementar navegación a las páginas de gestión de recursos
    console.log('Please delete excess resources before downgrading your plan.');
  }

  /**
   * Closes redirect error pop-up.
   *
   * Hides error pop-up, clears message and stops any
   * active window check interval.
   *
   * Related to:
   * - windowCheckInterval: Interval that checks if Stripe window closed
   *
   * @memberof PlanSettingsComponent
   */
  closeRedirectError(): void {
    this.showRedirectError = false;
    this.redirectErrorMessage = '';

    // Limpiar intervalo si existe
    if (this.windowCheckInterval) {
      clearInterval(this.windowCheckInterval);
      this.windowCheckInterval = null;
    }

    // Ocultar loading si está visible
    this.showRedirectLoading = false;
  }
}
```

Ø=ÜÁ features\account\components\profile-details

Ø=ÜÄ features\account\components\profile-details\profile-details.component.ts

```
import { Component, OnInit, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, FormGroup, Validators, ReactiveFormsModule } from '@angular/forms';
import { Store } from '@ngrx/store';
import { User } from '../../../overview/models/overview';
import { selectUser } from '../../../auth/store/user.selectios';
import { AuthService } from '../../../auth/service/authService';
import { updatePassword, EmailAuthProvider, reauthenticateWithCredential, deleteUser } from
'firebase/auth';
import { AccountDeletionService } from '../../../../shared/services/account-
deletion.service';
import { Router } from '@angular/router';
import { AppContextService } from '../../../../shared/context';
import { PasswordInputComponent } from '../../../../shared/components/password-input/
password-input.component';
/**
```

```
 * Component for managing user profile details.
 *
 * This component allows the user to:
 * - View and edit personal information (name, last name, email, phone, birthday)
 * - Change password with validation and reauthentication
 * - Delete account completely (Firebase data and authentication)
 *
 * Related to:
 * - AccountComponent: Displayed in "Profile Details" tab
 * - AuthService: Updates user data and gets current user
 * - AppContextService: Gets and updates user data in context
 * - AccountDeletionService: Deletes all user data from Firebase
 * - Store (NgRx): Updates user state in store
 * - Firebase Auth: Reauthentication and password change
 *
 * Main flow:
 * 1. On initialization, subscribes to user data from context
 * 2. Populates form with user data
 * 3. Allows updating profile (synchronizes with Firebase, context and store)
 * 4. Allows changing password (requires reauthentication)
 * 5. Allows deleting account (deletes Firebase data and authentication)
 *
 * @component
 * @selector app-profile-details
 * @standalone true
 */
@Component({
  selector: 'app-profile-details',
  imports: [CommonModule, ReactiveFormsModule, PasswordInputComponent],
  templateUrl: './profile-details.component.html',
  styleUrl: './profile-details.component.scss',
  standalone: true,
})
export class ProfileDetailsComponent implements OnInit {
  user: User | null = null;
  profileForm: FormGroup;
  passwordForm: FormGroup;
  isLoading = false;
  showPasswordForm = false;
  passwordChangeMessage = '';
  passwordChangeError = '';
  showDeleteModal = false;
  isDeletingAccount = false;
  deleteAccountError = '';

  // Inyectar servicios
  private authService = inject(AuthService);
  private fb = inject(FormBuilder);
  private accountDeletionService = inject(AccountDeletionService);
  private router = inject(Router);
  private appContext = inject(AppContextService);

  constructor(private store: Store) {
    this.profileForm = this.fb.group({
      firstName: ['', [Validators.required, Validators.minLength(2)]],
      lastName: ['', [Validators.required, Validators.minLength(2)]],
      email: ['', [Validators.required, Validators.email]],
      phoneNumber: [''],
      birthday: ['', [Validators.required]],
    });

    this.passwordForm = this.fb.group({
      currentPassword: ['', [Validators.required]],
      newPassword: ['', [Validators.required]], // Las validaciones específicas las maneja el componente
      confirmPassword: ['', [Validators.required]],
    }, { validators: this.passwordMatchValidator });
  }

  ngOnInit(): void {
    this.subscribeToContextData();
```

```
  }

  /**
   * Subscribes to user data from application context.
   *
   * When user data is received, populates the form
   * with current information.
   *
   * Related to:
   * - AppContextService.currentUser$: Observable of current user
   * - populateForm(): Populates form with user data
   *
   * @private
   * @memberof ProfileDetailsComponent
   */
  private subscribeToContextData(): void {
    // Suscribirse a los datos del usuario desde el contexto
    this.appContext.currentUser$.subscribe({
      next: (user) => {
        this.user = user;
        if (this.user) {
          this.populateForm();
        }
      },
      error: (err) => {
        console.error('Error fetching user data from context', err);
      },
    });
  }

  /**
   * Populates profile form with current user data.
   *
   * Updates all form fields with user values
   * (firstName, lastName, email, phoneNumber, birthday).
   *
   * @private
   * @memberof ProfileDetailsComponent
   */
  private populateForm(): void {
    if (this.user) {
      this.profileForm.patchValue({
        firstName: this.user.firstName || '',
        lastName: this.user.lastName || '',
        email: this.user.email || '',
        phoneNumber: this.user.phoneNumber || '',
        birthday: this.user.birthday || '',
      });
    }
  }

  /**
   * Custom validator to verify that passwords match.
   *
   * Compares newPassword and confirmPassword fields of the form.
   * If they don't match, sets an error on confirmPassword field.
   *
   * @private
   * @param form - FormGroup of password form
   * @returns null if passwords match, error object if they don't match
   * @memberof ProfileDetailsComponent
   */
  private passwordMatchValidator(form: FormGroup) {
    const newPassword = form.get('newPassword');
    const confirmPassword = form.get('confirmPassword');

    if (newPassword && confirmPassword && newPassword.value !== confirmPassword.value) {
      confirmPassword.setErrors({ passwordMismatch: true });
      return { passwordMismatch: true };
    }
```

```
155
156       return null;
157     }
158
159     /**
160      * Updates user profile with form data.
161      *
162      * This method synchronizes changes in three places:
163      * 1. Firebase: Updates data in database
164      * 2. AppContextService: Updates user in context (source of truth)
165      * 3. Store (NgRx): Updates user state in store
166      *
167      * Only updates firstName, lastName and birthday (email cannot be changed here).
168      *
169      * Related to:
170      * - AuthService.updateUser(): Updates data in Firebase
171      * - AppContextService.updateUserData(): Updates context
172      * - Store.dispatch(): Updates NgRx store
173      *
174      * @async
175      * @memberof ProfileDetailsComponent
176      */
177     async onUpdateProfile(): Promise<void> {
178       if (this.profileForm.valid && this.user) {
179         this.isLoading = true;
180         try {
181           const updatedData = {
182             firstName: this.profileForm.value.firstName,
183             lastName: this.profileForm.value.lastName,
184             birthday: this.profileForm.value.birthday,
185           };
186
187           // 1. Actualizar en Firebase
188           await this.authService.updateUser(this.user.id, updatedData);
189
190           // 2. Actualizar el usuario en el contexto (fuente de verdad)
191           this.appContext.updateUserData(updatedData);
192
193           // 3. Actualizar el usuario en el store
194           this.store.dispatch({
195             type: '[User] Update User',
196             user: {
197               ...this.user,
198               ...updatedData
199             }
200           });
201
202           console.log('  Profile updated successfully');
203         } catch (error) {
204           console.error(''L Error updating profile:', error);
205         } finally {
206           this.isLoading = false;
207         }
208       }
209     }
210
211     /**
212      * Changes user password.
213      *
214      * This method requires reauthentication for security before changing password.
215      * Performs:
216      * 1. Verifies user is authenticated
217      * 2. Reauthenticates user with current password
218      * 3. Updates password in Firebase Auth
219      * 4. Resets form and hides password form
220      *
221      * Handles specific errors:
222      * - auth/wrong-password: Current password incorrect
223      * - auth/weak-password: New password too weak
224      * - auth/requires-recent-login: Requires signing in again
```

```
   *
   * Related to:
   * - Firebase Auth: reauthenticateWithCredential, updatePassword
   * - AuthService.getCurrentUser(): Gets current user from Firebase
   *
   * @async
   * @memberof ProfileDetailsComponent
   */
  async onChangePassword(): Promise<void> {
    if (this.passwordForm.valid && this.user) {
      this.isLoading = true;
      this.passwordChangeMessage = '';
      this.passwordChangeError = '';

      try {
        const currentUser = this.authService.getCurrentUser();
        if (!currentUser) {
          throw new Error('User not authenticated');
        }

        const currentPassword = this.passwordForm.value.currentPassword;
        const newPassword = this.passwordForm.value.newPassword;

        // Reautenticar al usuario antes de cambiar la contraseña
        const credential = EmailAuthProvider.credential(
          this.user.email,
          currentPassword
        );

        await reauthenticateWithCredential(currentUser, credential);

        // Cambiar la contraseña
        await updatePassword(currentUser, newPassword);

        this.passwordChangeMessage = 'Password updated successfully';
        this.passwordForm.reset();
        this.showPasswordForm = false;

      } catch (error: any) {
        console.error(''L Error changing password:', error);

        if (error.code === 'auth/wrong-password') {
          this.passwordChangeError = 'Current password is incorrect';
        } else if (error.code === 'auth/weak-password') {
          this.passwordChangeError = 'New password is too weak';
        } else if (error.code === 'auth/requires-recent-login') {
          this.passwordChangeError = 'For security, please sign in again';
        } else {
          this.passwordChangeError = 'Error changing password. Please try again';
        }
      } finally {
        this.isLoading = false;
      }
    }
  }

  /**
   * Shows or hides password change form.
   *
   * When hidden, resets form and clears success and error messages.
   *
   * @memberof ProfileDetailsComponent
   */
  togglePasswordForm(): void {
    this.showPasswordForm = !this.showPasswordForm;
    this.passwordChangeMessage = '';
    this.passwordChangeError = '';
    this.passwordForm.reset();
  }

```

```
  /**
   * Gets error message for a profile form field.
   *
   * Returns specific error messages based on error type:
   * - required: Field required
   * - email: Invalid email
   * - minlength: Minimum length not reached
   *
   * Only shows errors if field has been touched.
   *
   * @param fieldName - Form field name
   * @returns Error message or empty string if no error
   * @memberof ProfileDetailsComponent
   */
  getFieldError(fieldName: string): string {
    const field = this.profileForm.get(fieldName);
    if (field?.errors && field.touched) {
      if (field.errors['required']) {
        return `${fieldName} is required`;
      }
      if (field.errors['email']) {
        return 'Invalid email';
      }
      if (field.errors['minlength']) {
        return `${fieldName} must have at least ${field.errors['minlength'].requiredLength}
characters`;
      }
    }
    return '';
  }

  /**
   * Gets error message for a password form field.
   *
   * Returns specific error messages based on error type:
   * - required: Field required
   * - minlength: Password must have at least 6 characters
   * - passwordMismatch: Passwords do not match
   *
   * Only shows errors if field has been touched.
   *
   * @param fieldName - Form field name
   * @returns Error message or empty string if no error
   * @memberof ProfileDetailsComponent
   */
  getPasswordFieldError(fieldName: string): string {
    const field = this.passwordForm.get(fieldName);
    if (field?.errors && field.touched) {
      if (field.errors['required']) {
        return `${fieldName} is required`;
      }
      if (field.errors['minlength']) {
        return 'Password must have at least 6 characters';
      }
      if (field.errors['passwordMismatch']) {
        return 'Passwords do not match';
      }
    }
    return '';
  }

  /**
   * Shows confirmation modal to delete account.
   *
   * This method opens the modal that requests confirmation before
   * proceeding with account deletion.
   *
   * @memberof ProfileDetailsComponent
   */
  showDeleteAccountModal(): void {
    this.showDeleteModal = true;
```

```
    this.deleteAccountError = '';
  }

  /**
   * Cancels account deletion process.
   *
   * Hides confirmation modal without performing any action.
   *
   * @memberof ProfileDetailsComponent
   */
  cancelDeleteAccount(): void {
    this.showDeleteModal = false;
    this.deleteAccountError = '';
  }

  /**
   * Deletes user account and all associated data.
   *
   * This method performs complete deletion in the following order:
   * 1. Deletes all user data from Firebase (AccountDeletionService)
   * 2. Deletes user from Firebase Authentication
   * 3. Clears local NgRx store
   * 4. Redirects user to login page
   *
   * Handles specific errors:
   * - auth/requires-recent-login: Requires signing in again for security
   * - auth/too-many-requests: Too many attempts, try again later
   *
   * Related to:
   * - AccountDeletionService.deleteUserData(): Deletes Firebase data
   * - Firebase Auth deleteUser(): Deletes user from authentication
   * - Store.dispatch(): Clears user state
   * - Router.navigate(): Redirects to login
   *
   * @async
   * @memberof ProfileDetailsComponent
   */
  async confirmDeleteAccount(): Promise<void> {
    if (!this.user) {
      this.deleteAccountError = 'User not found';
      return;
    }

    this.isDeletingAccount = true;
    this.deleteAccountError = '';

    try {

      // 1. Delete all Firebase data
      const firebaseDataDeleted: boolean = await
this.accountDeletionService.deleteUserData(this.user.id);
      if (!firebaseDataDeleted) {
        throw new Error('Error deleting Firebase data');
      }

      // 2. Delete user from Firebase Auth
      const currentUser = this.authService.getCurrentUser();
      if (currentUser) {
        await deleteUser(currentUser);
      }

      // 3. Clear local store
      this.store.dispatch({
        type: '[User] Clear User'
      });

      // 4. Redirect to login
      this.router.navigate(['/login']);

    } catch (error: any) {
```

```
      console.error(''L Error deleting account:', error);

      if (error.code === 'auth/requires-recent-login') {
        this.deleteAccountError = 'For security, you need to sign in again before deleting
your account.';
      } else if (error.code === 'auth/too-many-requests') {
        this.deleteAccountError = 'Too many attempts. Please try again later';
      } else {
        this.deleteAccountError = 'Error deleting account. Please try again';
      }
    } finally {
      this.isDeletingAccount = false;
    }
  }
}
```

Ø=ÜÁ features\account\components\subscription-history

Ø=ÜÄ features\account\components\subscription-history\subscription-history.component.ts

```typescript
import { Component, OnInit, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { Store } from '@ngrx/store';
import { User } from '../../../overview/models/overview';
import { selectUser } from '../../../auth/store/user.selectios';
import { SubscriptionService } from '../../../../shared/services/subscription-service';
import { PlanService } from '../../../../shared/services/planService';
import { Subscription } from '../../../../shared/services/subscription-service';
import { Plan } from '../../../../shared/services/planService';
import { UserStatus } from '../../../overview/models/overview';

/**
 * Component to display user subscription history.
 *
 * This component allows the user to:
 * - View subscription history
 * - Filter subscriptions by plan, date and search term
 * - Paginate results
 * - View details of each subscription (plan, status, date, amount)
 *
 * Related to:
 * - AccountComponent: Displayed in "Subscription History" tab (currently commented)
 * - SubscriptionService: Gets user subscriptions
 * - PlanService: Gets plan information to display names
 * - Store (NgRx): Gets current user
 *
 * Features:
 * - Search by subscription ID
 * - Filters by plan and date range
 * - Result pagination
 * - Date and currency formatting
 * - Mapping subscription statuses to CSS classes and readable text
 *
 * NOTE: This component is currently commented in AccountComponent template.
 *
 * @component
 * @selector app-subscription-history
 * @standalone true
 */
@Component({
  selector: 'app-subscription-history',
  imports: [CommonModule, FormsModule],
```

```
  templateUrl: './subscription-history.component.html',
  styleUrl: './subscription-history.component.scss',
  standalone: true,
})
export class SubscriptionHistoryComponent implements OnInit {
  user: User | null = null;
  subscriptions: Subscription[] = [];
  filteredSubscriptions: Subscription[] = [];
  plans: Plan[] = [];
  plansMap: { [key: string]: Plan } = {};

  // Pagination
  currentPage = 1;
  itemsPerPage = 10;
  totalPages = 0;

  // Filters
  searchTerm = '';
  showFilter = false;
  selectedPlan = '';
  dateFrom = '';
  dateTo = '';

  planNames: string[] = [];

  // Loading state
  isLoading = false;

  // Inyectar servicios
  private subscriptionService = inject(SubscriptionService);
  private planService = inject(PlanService);

  constructor(private store: Store) {}

  async ngOnInit(): Promise<void> {
    this.getUserData();
    await this.loadPlans();
  }

  /**
   * Gets current user data from NgRx store.
   *
   * Subscribes to selectUser selector and when user is obtained,
   * loads user subscriptions.
   *
   * Related to:
   * - Store.select(selectUser): NgRx selector to get user
   * - loadPayments(): Loads subscriptions when user exists
   *
   * @private
   * @memberof SubscriptionHistoryComponent
   */
  private getUserData(): void {
    this.store.select(selectUser).subscribe({
      next: (userData) => {
        this.user = userData.user;
        if (this.user) {
          this.loadPayments();
        }
      },
      error: (err) => {
        console.error('Error fetching user data', err);
      },
    });
  }

  /**
   * Loads all available plans from PlanService.
   *
   * This method:
```

```
     * 1. Gets all plans from PlanService
     * 2. Creates an array of plan names for filters
     * 3. Creates a plan map (by ID) for quick lookup
     *
     * Related to:
     * - PlanService.getAllPlans(): Gets all plans
     * - getPlanName(): Uses plansMap to get plan names
     *
     * @private
     * @async
     * @memberof SubscriptionHistoryComponent
     */
    private async loadPlans(): Promise<void> {
      try {
        this.plans = await this.planService.getAllPlans();
        this.planNames = this.plans.map(plan => plan.name);

        // Crear mapa de planes para búsqueda rápida por ID
        this.plansMap = {};
        this.plans.forEach(plan => {
          this.plansMap[plan.id] = plan;
        });
      } catch (error) {
        console.error('Error loading plans:', error);
      }
    }

    /**
     * Loads user subscriptions from SubscriptionService.
     *
     * Gets user's most recent subscription and displays it
     * in the list. Currently only loads one subscription (the most recent).
     *
     * Related to:
     * - SubscriptionService.getUserLatestSubscription(): Gets most recent subscription
     * - calculatePagination(): Calculates pagination after loading
     *
     * @private
     * @async
     * @memberof SubscriptionHistoryComponent
     */
    private async loadPayments(): Promise<void> {
      if (!this.user) return;

      this.isLoading = true;
      try {
        const latest = await this.subscriptionService.getUserLatestSubscription(this.user.id);
        this.subscriptions = latest ? [latest] : [];
        this.filteredSubscriptions = [...this.subscriptions];
        this.calculatePagination();
      } catch (error) {
        console.error('Error loading subscriptions:', error);
      } finally {
        this.isLoading = false;
      }
    }

    /**
     * Calculates total number of pages based on filtered results.
     *
     * If current page is greater than total pages,
     * resets to page 1.
     *
     * @private
     * @memberof SubscriptionHistoryComponent
     */
    private calculatePagination(): void {
      this.totalPages = Math.ceil(this.filteredSubscriptions.length / this.itemsPerPage);
      if (this.currentPage > this.totalPages) {
        this.currentPage = 1;
```

```
    }
  }

  /**
   * Getter that returns paginated subscriptions for current page.
   *
   * Calculates range of subscriptions to display based on:
   * - currentPage: Current page
   * - itemsPerPage: Number of items per page
   *
   * @returns Array of subscriptions for current page
   * @memberof SubscriptionHistoryComponent
   */
  get paginatedSubscriptions(): Subscription[] {
    const startIndex = (this.currentPage - 1) * this.itemsPerPage;
    const endIndex = startIndex + this.itemsPerPage;
    return this.filteredSubscriptions.slice(startIndex, endIndex);
  }

  /**
   * Executes when search term changes.
   *
   * Applies current filters (including new search term).
   *
   * @memberof SubscriptionHistoryComponent
   */
  onSearchChange(): void {
    this.applyFilters();
  }

  /**
   * Shows or hides filter panel.
   *
   * @memberof SubscriptionHistoryComponent
   */
  openFilter(): void {
    this.showFilter = !this.showFilter;
  }

  /**
   * Applies all active filters to subscriptions.
   *
   * This method filters subscriptions according to:
   * - Search term: Searches in subscription ID
   * - Date from: Filters subscriptions from a specific date
   * - Date to: Filters subscriptions until a specific date
   *
   * After filtering, recalculates pagination and resets to page 1.
   *
   * @memberof SubscriptionHistoryComponent
   */
  applyFilters(): void {
    let filtered = [...this.subscriptions];

    // Search filter
    if (this.searchTerm) {
      filtered = filtered.filter(subscription =>
        subscription.id?.toLowerCase().includes(this.searchTerm.toLowerCase())
      );
    }

    // Date filters
    if (this.dateFrom) {
      const fromDate = new Date(this.dateFrom);
      filtered = filtered.filter(subscription => {
        const subscriptionDate = subscription.created_at.toDate();
        return subscriptionDate >= fromDate;
      });
    }

```

```
    if (this.dateTo) {
      const toDate = new Date(this.dateTo);
      toDate.setHours(23, 59, 59, 999); // End of day
      filtered = filtered.filter(subscription => {
        const subscriptionDate = subscription.created_at.toDate();
        return subscriptionDate <= toDate;
      });
    }

    this.filteredSubscriptions = filtered;
    this.calculatePagination();
    this.currentPage = 1;
  }

  /**
   * Clears all active filters.
   *
   * Resets all filter fields (search, plan, dates)
   * and reapplies filters (showing all results).
   *
   * @memberof SubscriptionHistoryComponent
   */
  clearFilters(): void {
    this.searchTerm = '';
    this.selectedPlan = '';
    this.dateFrom = '';
    this.dateTo = '';
    this.applyFilters();
  }

  /**
   * Navigates to previous page.
   *
   * Only navigates if there's a previous page (currentPage > 1).
   *
   * @memberof SubscriptionHistoryComponent
   */
  prevPage(): void {
    if (this.currentPage > 1) {
      this.currentPage--;
    }
  }

  /**
   * Navigates to next page.
   *
   * Only navigates if there's a next page (currentPage < totalPages).
   *
   * @memberof SubscriptionHistoryComponent
   */
  nextPage(): void {
    if (this.currentPage < this.totalPages) {
      this.currentPage++;
    }
  }

  /**
   * Navigates directly to a specific page.
   *
   * Validates that page is within valid range (1 to totalPages).
   *
   * @param page - Page number to navigate to
   * @memberof SubscriptionHistoryComponent
   */
  goToPage(page: number): void {
    if (page >= 1 && page <= this.totalPages) {
      this.currentPage = page;
    }
  }

```

```
  /**
   * Gets plan name by its ID.
   *
   * Searches for plan in plan map (plansMap) and returns its name.
   * If not found, returns "Unknown Plan".
   *
   * Related to:
   * - plansMap: Plan map created in loadPlans()
   *
   * @param planId - Plan ID
   * @returns Plan name or "Unknown Plan" if not found
   * @memberof SubscriptionHistoryComponent
   */
  getPlanName(planId: string): string {
    const plan = this.plansMap[planId];
    return plan ? plan.name : 'Unknown Plan';
  }



  /**
   * Converts subscription status to readable text for display.
   *
   * Maps UserStatus states to more friendly text:
   * - PURCHASED !’ "Completed"
   * - PENDING !’ "Pending"
   * - PROCESSING !’ "Processing"
   * - CANCELLED !’ "Cancelled"
   * - EXPIRED !’ "Expired"
   * - BANNED !’ "Banned"
   * - ADMIN !’ "Admin"
   * - CREATED !’ "Created"
   *
   * @param status - Subscription status (UserStatus)
   * @returns Readable status text
   * @memberof SubscriptionHistoryComponent
   */
  getStatusDisplay(status: string): string {
    const statusMap: { [key: string]: string } = {
      [UserStatus.PURCHASED]: 'Completed',
      [UserStatus.PENDING]: 'Pending',
      [UserStatus.PROCESSING]: 'Processing',
      [UserStatus.CANCELLED]: 'Cancelled',
      [UserStatus.EXPIRED]: 'Expired',
      [UserStatus.BANNED]: 'Banned',
      [UserStatus.ADMIN]: 'Admin',
      [UserStatus.CREATED]: 'Created'
    };
    return statusMap[status] || status;
  }

  /**
   * Gets CSS class corresponding to subscription status.
   *
   * Maps UserStatus states to CSS classes for styling:
   * - PURCHASED !’ "completed"
   * - PENDING !’ "pending"
   * - PROCESSING !’ "processing"
   * - CANCELLED !’ "cancelled"
   * - EXPIRED !’ "expired"
   * - BANNED !’ "banned"
   * - ADMIN !’ "admin"
   * - CREATED !’ "created"
   *
   * @param status - Subscription status (UserStatus)
   * @returns CSS class name
   * @memberof SubscriptionHistoryComponent
   */
  getStatusClass(status: string): string {
    const statusClassMap: { [key: string]: string } = {
```

```
      [UserStatus.PURCHASED]: 'completed',
      [UserStatus.PENDING]: 'pending',
      [UserStatus.PROCESSING]: 'processing',
      [UserStatus.CANCELLED]: 'cancelled',
      [UserStatus.EXPIRED]: 'expired',
      [UserStatus.BANNED]: 'banned',
      [UserStatus.ADMIN]: 'admin',
      [UserStatus.CREATED]: 'created'
    };
    return statusClassMap[status] || 'unknown';
  }

  /**
   * Formats a date for display in the interface.
   *
   * Handles both Firebase Timestamp objects and Date objects.
   * Formats date in readable format (e.g.: "Jan 15, 2024, 10:30 AM").
   *
   * @param date - Date to format (can be Firebase Timestamp or Date)
   * @returns Formatted date or "N/A" if no date
   * @memberof SubscriptionHistoryComponent
   */
  formatDate(date: any): string {
    if (!date) return 'N/A';
    const dateObj = date.toDate ? date.toDate() : new Date(date);
    return dateObj.toLocaleDateString('en-US', {
      year: 'numeric',
      month: 'short',
      day: 'numeric',
      hour: '2-digit',
      minute: '2-digit'
    });
  }

  /**
   * Formats an amount as currency according to provided currency code.
   *
   * Uses Intl.NumberFormat to format amount with corresponding
   * currency symbol.
   *
   * @param amount - Amount to format
   * @param currency - Currency code (e.g.: "usd", "eur")
   * @returns Amount formatted as currency (e.g.: "$99.00")
   * @memberof SubscriptionHistoryComponent
   */
  formatCurrency(amount: number, currency: string): string {
    return new Intl.NumberFormat('en-US', {
      style: 'currency',
      currency: currency.toUpperCase()
    }).format(amount);
  }
}
```

Ø=ÜÁ features\account\mocks

Ø=ÜÄ features\account\mocks\account-mocks.ts

```
import { PlanCard, PlanDetails } from '../models/account-settings';

/**
 * Mock data for user plan details.
 *
```

```
 * This object simulates the current user's plan information.
 * Currently used in AccountComponent to initialize plan data.
 *
 * NOTE: In production, this data should be obtained from the subscription service.
 *
 * @constant MOCK_PLAN_DETAILS
 * @type {PlanDetails}
 * @see PlanDetails
 */
export const MOCK_PLAN_DETAILS: PlanDetails = {
  currentPlan: 'Pro Plan',
  renewalDate: '2025-08-27',
  remainingUntilRenewal: '16 days',
  price: 250.0,
  activationFee: null,
  billingCycle: 'Monthly',
};

/**
 * Array of available plans for subscription.
 *
 * This array contains the definition of all plans displayed
 * in the plan selection interface. Each plan includes:
 * - Price and period information
 * - Features and limits (trading accounts, strategies, etc.)
 * - Visual information (icons, colors)
 * - Action button text (CTA)
 *
 * Used in:
 * - PlanSettingsComponent: As initial data before loading from service
 *
 * NOTE: In production, this data should be obtained from PlanService.
 *
 * @constant PLANS
 * @type {PlanCard[]}
 * @see PlanCard
 */
export const PLANS: PlanCard[] = [
  {
    name: 'Free',
    price: 0,
    period: '/month',
    icon: 'circle',
    color: '#4b7ee8',
    features: [
      { label: 'Trading Accounts', value: '1' },
      { label: 'Consistency Rules', value: 'YES' },
      { label: 'Trading Journal', value: 'YES' },
      { label: 'Live Statistics', value: 'YES' },
    ],
    cta: 'Change Plan',
  },
  {
    name: 'Starter',
    price: 35,
    period: '/month',
    icon: 'circle',
    color: '#4b7ee8',
    features: [
      { label: 'Trading Accounts', value: '2' },
      { label: 'Consistency Rules', value: 'YES' },
      { label: 'Trading Journal', value: 'YES' },
      { label: 'Live Statistics', value: 'YES' },
    ],
    cta: 'Change Plan',
  },
  {
    name: 'Pro',
    price: 99,
    period: '/month',
```

```
    mostPopular: true,
    icon: 'square',
    color: '#d1ff81',
    features: [
      { label: 'Trading Accounts', value: '6' },
      { label: 'Consistency Rules', value: 'YES' },
      { label: 'Trading Journal', value: 'YES' },
      { label: 'Live Statistics', value: 'YES' },
    ],
    cta: 'Get Starter Now',
  },
];

```

Ø=ÜÁ features\account\models

Ø=ÜÄ features\account\models\account-settings.ts

```typescript
/**
 * Interface that represents the details of the user's current plan.
 *
 * This interface contains all information related to the user's active
 * subscription plan, including billing and renewal information.
 *
 * Used in:
 * - AccountComponent: To pass plan data to child component
 * - PlanSettingsComponent: To display current plan information
 *
 * @interface PlanDetails
 */
export interface PlanDetails {
  currentPlan: string;
  renewalDate: string;
  remainingUntilRenewal: string;
  price: number;
  activationFee: string | null;
  billingCycle: string;
}

/**
 * Interface that represents a plan card in the user interface.
 *
 * This interface defines the data structure needed to display a plan
 * in the plan comparison and selection interface. Includes visual
 * information (icons, colors) and functional information (price, features, CTA).
 *
 * Used in:
 * - PlanSettingsComponent: To build and display available plan cards
 * - account-mocks.ts: To define mock data for plans
 *
 * @interface PlanCard
 */
export interface PlanCard {
  name: string;
  price: number;
  period: string;
  mostPopular?: boolean;
  icon: string;
  color: string;
  features: {
    label: string;
    value: string;
  }[];
```

```
  cta: string;
}
```

Ø=ÜÁ features\add-account

Ø=ÜÄ features\add-account\add-account.component.ts

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import {
  FormBuilder,
  FormGroup,
  ReactiveFormsModule,
  Validators,
} from '@angular/forms';
import {
  PhoneInputComponent,
  TextInputComponent,
} from '../../shared/components';
import { Router, RouterLink } from '@angular/router';
import { PasswordInputComponent } from '../../shared/components/password-input/password-input.component';
import { AuthService } from '../auth/service/authService';
import { Store } from '@ngrx/store';
import { selectUser } from '../auth/store/user.selectios';
import { AccountData } from '../auth/models/userModel';
import { Timestamp } from 'firebase/firestore';
import { first } from 'rxjs';

/**
 * Component for adding a new trading account.
 *
 * This component provides a form interface for users to register
 * a new trading account with their broker. It collects account
 * information including email, password, broker details, and account
 * identification data.
 *
 * Related to:
 * - AuthService: Creates the account in Firebase
 * - Store (NgRx): Gets current user data
 * - Router: Navigates to trading accounts page after creation
 *
 * @component
 * @selector app-add-account
 * @standalone true
 */
@Component({
  selector: 'app-add-account',
  standalone: true,
  imports: [
    CommonModule,
    ReactiveFormsModule,
    TextInputComponent,
    PasswordInputComponent,
    RouterLink,
  ],
  templateUrl: './add-account.component.html',
  styleUrls: ['./add-account.component.scss'],
})
export class AddAccountComponent {
  /** Form group containing all trading account input fields */
  accountForm: FormGroup;

```

```
  /**
   * Constructor for AddAccountComponent.
   *
   * Initializes the reactive form with all required fields and validators:
   * - emailTradingAccount: Required email validation
   * - brokerPassword: Required, minimum 6 characters
   * - broker: Required broker name
   * - server: Required server name
   * - accountName: Required account name
   * - accountID: Required account ID
   * - accountNumber: Required, numeric pattern only
   *
   * @param fb - FormBuilder for creating reactive forms
   * @param authService - Service for authentication and account operations
   * @param router - Router for navigation
   * @param store - NgRx Store for accessing user state
   */
  constructor(
    private fb: FormBuilder,
    private authService: AuthService,
    private router: Router,
    private store: Store
  ) {
    this.accountForm = this.fb.group({
      emailTradingAccount: ['', [Validators.required, Validators.email]],
      brokerPassword: ['', [Validators.required, Validators.minLength(6)]],
      broker: ['', [Validators.required]],
      server: ['', [Validators.required]],
      accountName: ['', [Validators.required]],
      accountID: ['', [Validators.required]],
      accountNumber: [
        '',
        [Validators.required, Validators.pattern('^[0-9]*$')],
      ],
    });
  }

  /**
   * Handles form submission when user clicks the submit button.
   *
   * Validates the form and either:
   * - Processes registration if form is valid
   * - Marks all form fields as touched to show validation errors if invalid
   *
   * @memberof AddAccountComponent
   */
  onSubmit(): void {
    if (this.accountForm.valid) {
      this.processRegistration();
    } else {
      this.markFormGroupTouched();
    }
  }

  /**
   * Processes the account registration.
   *
   * Gets the current user from the store, creates an account object
   * with the form data, and saves it to Firebase. After successful
   * creation, navigates to the trading accounts page.
   *
   * Related to:
   * - Store.select(selectUser): Gets current user from NgRx store
   * - AuthService.createAccount(): Creates account in Firebase
   * - createAccountObject(): Builds account data object
   * - Router.navigate(): Redirects to trading accounts page
   *
   * @private
   * @memberof AddAccountComponent
   */
```

```
  private processRegistration(): void {
    this.store
      .select(selectUser)
      .pipe(first())
      .subscribe((user) => {
        const userId = user?.user?.id || '';
        this.authService.createAccount(this.createAccountObject(userId));
        this.router.navigate(['/trading-accounts']);
      });
  }

  /**
   * Marks all form controls as touched to trigger validation error display.
   *
   * This method iterates through all form controls and marks them as touched,
   * which causes Angular to display validation error messages for invalid fields.
   *
   * @private
   * @memberof AddAccountComponent
   */
  private markFormGroupTouched(): void {
    Object.keys(this.accountForm.controls).forEach((key) => {
      const control = this.accountForm.get(key);
      control?.markAsTouched();
    });
  }

  /**
   * Creates an AccountData object from form values.
   *
   * Generates a unique ID for the account using timestamp and random string,
   * then constructs an AccountData object with all form values and metadata.
   *
   * The unique ID is generated using:
   * - Current timestamp in base36 format
   * - Random string (6 characters)
   * - Format: `id_{timestamp}_{random}`
   *
   * @private
   * @param id - User ID to associate with the account
   * @returns AccountData object ready to be saved to Firebase
   * @memberof AddAccountComponent
   */
  private createAccountObject(id: string): AccountData {
    const timestamp = Date.now().toString(36);
    const randomPart = Math.random().toString(36).substring(2, 8);
    const uniqueId = `id_${timestamp}_${randomPart}`;
    return {
      id: uniqueId,
      userId: id,
      emailTradingAccount: this.accountForm.value.emailTradingAccount,
      brokerPassword: this.accountForm.value.brokerPassword,
      broker: this.accountForm.value.broker,
      server: this.accountForm.value.server,
      accountName: this.accountForm.value.accountName,
      accountID: this.accountForm.value.accountID,
      accountNumber: Number(this.accountForm.value.accountNumber),
      createdAt: Timestamp.now(),
    };
  }
}

```

Ø=ÜÁ features\auth\login

Ø=ÜÄ features\auth\login\login.ts

```ts
import { Component } from '@angular/core';
import { NgIf } from '@angular/common';
import {
  FormBuilder,
  FormGroup,
  ReactiveFormsModule,
  Validators,
  AbstractControl,
  ValidationErrors,
} from '@angular/forms';
import { PasswordInputComponent } from '../../../shared/components/password-input/password-input.component';
import { TextInputComponent } from '../../../shared/components';
import { AuthService } from '../../../shared/services/auth.service';
import { Router, RouterLink } from '@angular/router';
import { Store } from '@ngrx/store';
import { setUserData } from '../store/user.actions';
import { User } from '../../overview/models/overview';
import { UserCredentials } from '../models/userModel';
import { AppContextService } from '../../../shared/context';
import { AlertService } from '../../../shared/services/alert.service';
import { ForgotPasswordPopupComponent } from '../../../shared/pop-ups/forgot-password/forgot-password.component';
@Component({
  selector: 'app-login',
  standalone: true,
  imports: [
    ReactiveFormsModule,
    PasswordInputComponent,
    TextInputComponent,
    RouterLink,
    ForgotPasswordPopupComponent,
  ],
  templateUrl: './login.html',
  styleUrl: './login.scss',
})
export class Login {
  loginForm: FormGroup;
  showPassword = false;
  forgotVisible = false;

  constructor(
    private fb: FormBuilder,
    private authService: AuthService,
    private store: Store,
    private router: Router,
    private appContext: AppContextService,
    private alertService: AlertService
  ) {
    this.loginForm = this.fb.group({
      loginEmail: ['', [Validators.required, Validators.email, this.emailValidator]],
      password: ['', [Validators.required]],
      rememberMe: [false],
    });

  }

  onSubmit(): void {
    // Validar campos antes de proceder
    this.validateLoginFields();

    if (this.loginForm.valid) {
```

```
      const userCredentials = this.createUserCredentialsObject();

      // Establecer estado de carga
      this.appContext.setLoading('user', true);
      this.appContext.setError('user', null);

      this.authService
        .login(userCredentials)
        .then((response: any) => {
          this.authService
            .getUserData(response.user.uid)
            .then((userData: User) => {
              // Actualizar contexto con datos del usuario
              this.appContext.setCurrentUser(userData);

              // Mantener compatibilidad con NgRx
              this.store.dispatch(setUserData({ user: userData }));

              // Limpiar estado de carga
              this.appContext.setLoading('user', false);

              // Navegar según el tipo de usuario
              if (userData.isAdmin) {
                this.router.navigate(['/overview']);
              } else {
                this.router.navigate(['/strategy']);
              }
            })
            .catch((error: any) => {
              this.appContext.setLoading('user', false);
              this.appContext.setError('user', 'Error al obtener datos del usuario');
              this.handleLoginError(error);
            });
        })
        .catch((error: any) => {
          this.appContext.setLoading('user', false);
          this.appContext.setError('user', 'Error de autenticación');
          this.handleLoginError(error);
        });
    }
  }

  openForgot(): void {
    this.forgotVisible = true;
  }

  closeForgot(): void {
    this.forgotVisible = false;
  }

  private createUserCredentialsObject(): UserCredentials {
    return {
      email: this.loginForm.value.loginEmail,
      password: this.loginForm.value.password,
    };
  }

  togglePasswordVisibility(): void {
    this.showPassword = !this.showPassword;
  }

  signInWithGoogle(): void {
    // TODO: Implement Google sign-in
    console.log('Google sign-in');
  }

  signInWithApple(): void {
    // TODO: Implement Apple sign-in
    console.log('Apple sign-in');
  }
```

```
132
133   // Validador personalizado para email
134   private emailValidator(control: AbstractControl): ValidationErrors | null {
135     if (!control.value) return null;
136
137     const emailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
138
139     if (!emailRegex.test(control.value)) {
140       return { invalidEmailFormat: true };
141     }
142
143     return null;
144   }
145
146   // Validar campos del formulario
147   private validateLoginFields(): void {
148     const errors: string[] = [];
149
150     const emailControl = this.loginForm.get('loginEmail');
151     const passwordControl = this.loginForm.get('password');
152
153     // Validar email
154     if (emailControl?.errors?.['required']) {
155       errors.push('Email is required');
156     } else if (emailControl?.errors?.['email']) {
157       errors.push('Invalid email format');
158     } else if (emailControl?.errors?.['invalidEmailFormat']) {
159       errors.push('Invalid email format. Must contain @ and a valid domain');
160     }
161
162     // Validar contraseña
163     if (passwordControl?.errors?.['required']) {
164       errors.push('Password is required');
165     }
166
167     // Mostrar alerta si hay errores
168     if (errors.length > 0) {
169       this.alertService.showError('Validation errors:\n\n' + errors.join('\n'), 'Validation
Error');
170     }
171   }
172
173   // Manejar errores de login
174   private handleLoginError(error: any): void {
175     console.error('Login error:', error);
176
177     let errorMessage = 'Login failed. ';
178
179     // Verificar tipo de error específico
180     if (error.code === 'auth/user-not-found') {
181       errorMessage += 'No account found with this email.';
182     } else if (error.code === 'auth/wrong-password') {
183       errorMessage += 'Incorrect password.';
184     } else if (error.code === 'auth/invalid-email') {
185       errorMessage += 'Invalid email format.';
186     } else if (error.code === 'auth/user-disabled') {
187       errorMessage += 'This account has been disabled.';
188     } else if (error.code === 'auth/too-many-requests') {
189       errorMessage += 'Too many failed attempts. Please try again later.';
190     } else {
191       // Error genérico - verificar si es problema de credenciales
192       errorMessage += 'Email or password is incorrect. Please check your credentials and try
again.';
193     }
194
195     this.alertService.showError(errorMessage, 'Login Error');
196   }
197 }
198
```

Ø=ÜÁ features\auth\models

Ø=ÜÄ features\auth\models\linkModels.ts

```

interface LinkToken {
  id: string;
  userId: string;
}
```

Ø=ÜÄ features\auth\models\userModel.ts

```
import { Timestamp } from 'firebase/firestore';

export interface UserCredentials {
  email: string;
  password: string;
}

export interface AccountData {
  id: string;
  userId: string;
  emailTradingAccount: string;
  brokerPassword: string;
  broker: string;
  server: string;
  accountName: string;
  accountID: string;
  accountNumber: number;
  initialBalance?: number;
  balance?: number;
  netPnl?: number;
  profit?: number;
  bestTrade?: number;
  createdAt: Timestamp;
}

```

Ø=ÜÁ features\auth\service

Ø=ÜÄ features\auth\service\authService.ts

```
// Deliberadamente mantener esta ruta para compatibilidad.
// Reexporta el nuevo servicio centralizado en shared/services.
export { AuthService } from '../../../shared/services/auth.service';

```

Ø=ÜÁ features\auth\signup

Ø=ÜÄ features\auth\signup\signup.ts

```typescript
import { Component, OnInit } from '@angular/core';
import { Router, RouterLink } from '@angular/router';
import { CommonModule } from '@angular/common';
import {
  FormBuilder,
  FormGroup,
  Validators,
  ReactiveFormsModule,
  AbstractControl,
  ValidationErrors,
} from '@angular/forms';
import { PhoneInputComponent } from '../../../shared/components/phone-input/phone-input.component';
import { BirthdayInputComponent } from '../../../shared/components/birthday-input/birthday-input.component';
import { TextInputComponent } from '../../../shared/components/text-input/text-input.component';
import { AuthService } from '../../../shared/services/auth.service';
import { PasswordInputComponent } from '../../../shared/components/password-input/password-input.component';
import { User, UserStatus } from '../../overview/models/overview';
import { UserCredentials } from '../models/userModel';
import { Plan, PlanService } from '../../../shared/services/planService';
import { PlanSelectionComponent, PlanCard } from './components/plan-selection/plan-selection.component';
import { Subscription, SubscriptionService } from '../../../shared/services/subscription-service';
import { setUserData } from '../store/user.actions';
import { Store } from '@ngrx/store';
import { AppContextService } from '../../../shared/context';
import { StripeLoaderPopupComponent } from '../../../shared/pop-ups/stripe-loader-popup/stripe-loader-popup.component';
import { AlertService } from '../../../shared/services/alert.service';

@Component({
  selector: 'app-signup',
  standalone: true,
  imports: [
    CommonModule,
    ReactiveFormsModule,
    PhoneInputComponent,
    BirthdayInputComponent,
    TextInputComponent,
    PasswordInputComponent,
    RouterLink,
    PlanSelectionComponent,
    StripeLoaderPopupComponent,
  ],
  templateUrl: './signup.html',
  styleUrl: './signup.scss',
})
export class SignupComponent implements OnInit {
  signupForm: FormGroup;
  accountForm: FormGroup;
  currentStep = 1;
  isAdminSignup: boolean = false;
  showPlanSelection = false;
  userData: any = null;
  selectedPlan: PlanCard | null = null;
  currentUserId: string = '';

  // Estados para loader y error de Stripe
  showStripeLoader = false;
  showStripeError = false;
  stripeErrorMessage = '';

  constructor(
    private fb: FormBuilder,
```

```
    private authService: AuthService,
    private router: Router,
    private planService: PlanService,
    private subscriptionService: SubscriptionService,
    private store: Store,
    private appContext: AppContextService,
    private alertService: AlertService
  ) {
    this.signupForm = this.fb.group({
      firstName: ['', [Validators.required, Validators.minLength(2)]],
      lastName: ['', [Validators.required, Validators.minLength(2)]],
      phoneNumber: ['', [Validators.required, this.phoneValidator]],
      birthday: ['', [Validators.required, this.ageValidator]],
      email: ['', [Validators.required, Validators.email, this.emailValidator]],
      password: ['', [Validators.required, Validators.minLength(6)]],
    });

    this.accountForm = this.fb.group({
      emailTradingAccount: ['', [Validators.required, Validators.email]],
      brokerPassword: ['', [Validators.required, Validators.minLength(6)]],
      server: ['', [Validators.required]],
      accountName: ['', [Validators.required]],
      accountID: ['', [Validators.required]],
      accountNumber: [
        '',
        [Validators.required, Validators.pattern('^[0-9]*$')],
      ],
    });
  }

  ngOnInit(): void {
    const currentUrl = this.router.url;
    if (currentUrl === '/admin-signup') {
      this.isAdminSignup = true;
    }
  }

  onChange(): void {
    console.log('Form changed:', this.signupForm.value);
  }
  async onSubmit(): Promise<void> {
    if (this.signupForm.valid) {
      try {
        // Establecer estado de carga
        this.appContext.setLoading('user', true);
        this.appContext.setError('user', null);

        // Crear credenciales del usuario
        const userCredentials = this.createUserCredentialsObject();

        // Verificar que el email no esté ya registrado
        const existingUser = await this.authService.getUserByEmail(userCredentials.email);

        if (existingUser) {
          this.alertService.showError('This email is already registered. Please use a
different email or try logging in.', 'Email Already Registered');
          this.appContext.setLoading('user', false);
          return;
        }

        // Crear el usuario en Firebase Auth
        const userResponse = await this.authService.register(userCredentials);
        const userId = userResponse.user.uid;

        // Crear el token y objeto usuario
        const token = this.createTokenObject(userId);
        const user: User = await this.createUserObject(userId, token.id);

        // Configurar como admin si corresponde
        if (this.isAdminSignup) {
          user.isAdmin = true;
```

```
          user.status = UserStatus.ADMIN;
        } else {
          user.status = UserStatus.ACTIVE;
        }

        // Crear usuario y token en Firestore
        await this.authService.createUser(user as User);
        await this.authService.createLinkToken(token);

        // Crear suscripción gratuita activa por defecto
        const freeSubscriptionData: Omit<Subscription, 'id' | 'created_at' | 'updated_at'> =
{         planId: "Cb1B0tpxdE6AP6eMZDo0",
          status: UserStatus.ACTIVE,
          userId: userId,
        };

        await this.subscriptionService.createSubscription(userId, freeSubscriptionData);
        // Iniciar sesión automáticamente
        const loginResponse = await this.authService.login(userCredentials);
        const userData = await this.authService.getUserData(loginResponse.user.uid);

        // Actualizar contexto con datos completos del usuario
        this.appContext.setCurrentUser(userData);

        // Mantener compatibilidad con NgRx
        this.store.dispatch(setUserData({ user: userData }));

        // Guardar userId para usar en la selección de plan
        this.currentUserId = userId;

        // Limpiar estado de carga
        this.appContext.setLoading('user', false);

        // Guardar datos del usuario para mostrar en la selección de planes
        this.userData = this.signupForm.value;

        // Si es admin, ir directo al dashboard
        if (userData.isAdmin) {
          this.router.navigate(['/overview']);
        } else {
          // Usuario normal: mostrar selección de planes
          this.showPlanSelection = true;
        }

      } catch (error: any) {
        this.appContext.setLoading('user', false);
        this.appContext.setError('user', 'Error during registration');

        const errorMessage = error.message || 'An error occurred during registration. Please
try again.';
        this.alertService.showError(errorMessage, 'Registration Error');

        this.handleRegistrationError(error);
      }
    } else {
      this.showValidationErrors();
    }
  }

  private markFormGroupTouched(): void {
    Object.keys(this.signupForm.controls).forEach((key) => {
      const control = this.signupForm.get(key);
      control?.markAsTouched();
    });
  }

  signInWithGoogle(): void {
    console.log('Sign in with Google');
  }

  signInWithApple(): void {
```

```
    console.log('Sign in with Apple');
  }

  private createUserCredentialsObject(): UserCredentials {
    return {
      email: this.signupForm.value.email,
      password: this.signupForm.value.password,
    };
  }

  private async createUserObject(id: string, tokenId: string): Promise<User> {

    return {
      id: id,
      email: this.signupForm.value.email,
      tokenId: tokenId,
      firstName: this.signupForm.value.firstName,
      lastName: this.signupForm.value.lastName,
      phoneNumber: this.signupForm.value.phoneNumber,
      birthday: this.signupForm.value.birthday,
      best_trade: 0,
      netPnl: 0,
      number_trades: 0,
      profit: 0,
      status: UserStatus.CREATED,
      strategy_followed: 0,
      subscription_date: new Date().getTime(),
      lastUpdated: new Date().getTime(),
      total_spend: 0,
      isAdmin: false,
      trading_accounts: 0,
      strategies: 0,
    };
  }

  private createTokenObject(userId: string): LinkToken {
    return {
      id: this.signupForm.value.email.split('@')[0] + userId.substring(0, 4),
      userId: userId,
    };
  }

  // Validadores personalizados
  private phoneValidator(control: AbstractControl): ValidationErrors | null {
    if (!control.value) return null;

    const phoneRegex = /^[\+]?[1-9][\d]{0,15}$/;
    const cleanPhone = control.value.replace(/[\s\-\(\)]/g, '');

    if (!phoneRegex.test(cleanPhone)) {
      return { invalidPhone: true };
    }

    if (cleanPhone.length < 10 || cleanPhone.length > 15) {
      return { invalidPhoneLength: true };
    }

    return null;
  }

  private ageValidator(control: AbstractControl): ValidationErrors | null {
    if (!control.value) return null;

    const today = new Date();
    const birthDate = new Date(control.value);
    let age = today.getFullYear() - birthDate.getFullYear();
    const monthDiff = today.getMonth() - birthDate.getMonth();

    if (monthDiff < 0 || (monthDiff === 0 && today.getDate() < birthDate.getDate())) {
      age--;
```

```
    }

    if (age < 18) {
      return { underage: true };
    }

    return null;
  }

  private emailValidator(control: AbstractControl): ValidationErrors | null {
    if (!control.value) return null;

    const emailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

    if (!emailRegex.test(control.value)) {
      return { invalidEmailFormat: true };
    }

    return null;
  }

  private showValidationErrors(): void {
    const errors: string[] = [];

    // Verificar errores específicos de cada campo
    const firstNameControl = this.signupForm.get('firstName');
    const lastNameControl = this.signupForm.get('lastName');
    const emailControl = this.signupForm.get('email');
    const phoneControl = this.signupForm.get('phoneNumber');
    const birthdayControl = this.signupForm.get('birthday');
    const passwordControl = this.signupForm.get('password');

    if (firstNameControl?.errors?.['required']) {
      errors.push('First name is required');
    } else if (firstNameControl?.errors?.['minlength']) {
      errors.push('First name must be at least 2 characters');
    }

    if (lastNameControl?.errors?.['required']) {
      errors.push('Last name is required');
    } else if (lastNameControl?.errors?.['minlength']) {
      errors.push('Last name must be at least 2 characters');
    }

    if (emailControl?.errors?.['required']) {
      errors.push('Email is required');
    } else if (emailControl?.errors?.['email']) {
      errors.push('Invalid email format');
    } else if (emailControl?.errors?.['invalidEmailFormat']) {
      errors.push('Invalid email format. Must contain @ and a valid domain');
    }

    if (phoneControl?.errors?.['required']) {
      errors.push('Phone number is required');
    } else if (phoneControl?.errors?.['invalidPhone']) {
      errors.push('Invalid phone number format');
    } else if (phoneControl?.errors?.['invalidPhoneLength']) {
      errors.push('Phone number must be between 10 and 15 digits');
    }

    if (birthdayControl?.errors?.['required']) {
      errors.push('Birthday is required');
    } else if (birthdayControl?.errors?.['underage']) {
      errors.push('You must be 18 years or older to register');
    }

    if (passwordControl?.errors?.['required']) {
      errors.push('Password is required');
    } else if (passwordControl?.errors?.['minlength']) {
      errors.push('Password must be at least 6 characters');
```

```
    }

    // Mostrar alerta con todos los errores
    if (errors.length > 0) {
      this.alertService.showError('Validation errors:\n\n' + errors.join('\n'), 'Validation Error');
    }

    this.markFormGroupTouched();
  }

  async onPlanSelected(plan: PlanCard): Promise<void> {
    this.selectedPlan = plan;

    // Si selecciona el plan Free, redirigir al dashboard (sin loader ni pop-ups de Stripe)
    if (plan.name.toLowerCase() === 'free') {
      this.router.navigate(['/strategy']);
      return;
    }

    // Solo para planes de pago: mostrar loader y manejar errores de Stripe
    try {
      this.showStripeLoader = true;

      // Variable para controlar si hay error
      let hasError = false;
      let errorMessage = '';

      try {
        await this.createCheckoutSession(plan.name);
      } catch (error) {
        // Marcar que hay error pero no mostrar pop-up aún
        hasError = true;
        errorMessage = 'Error redirecting to payment. Please try again.';
        console.error('Error during checkout session creation:', error);
      }

      // Esperar mínimo 2 segundos antes de mostrar error o ocultar loader
      setTimeout(() => {
        if (hasError) {
          // Si hay error, mostrar pop-up de error
          this.showStripeLoader = false;
          this.showStripeError = true;
          this.stripeErrorMessage = errorMessage;
        } else {
          // Si no hay error, el loader se ocultará automáticamente por la redirección
          this.showStripeLoader = false;
        }
      }, 2000);

    } catch (error: any) {
      console.error(''L Error in plan selection:', error);
      this.showStripeLoader = false;
      this.showStripeError = true;
      this.stripeErrorMessage = 'Error processing your plan selection. Please try again.';
    }
  }

  private async createCheckoutSession(planName: string): Promise<void> {
    try {
      // Obtener el plan completo desde el servicio
      const plans = await this.planService.searchPlansByName(planName);
      const selectedPlan = plans && plans.length > 0 ? plans[0] : null;

      if (!selectedPlan || !selectedPlan.planPriceId) {
        throw new Error('Plan price ID not found');
      }

      // Obtener el token de Firebase
      const bearerTokenFirebase = await
this.authService.getBearerTokenFirebase(this.currentUserId);
```

```
      // Crear checkout session
      const response = await fetch('https://api.tradeswitch.io/payments/create-checkout-session', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Bearer ${bearerTokenFirebase}`
        },
        body: JSON.stringify({
          priceId: selectedPlan.planPriceId,
        })
      });

      if (!response.ok) {
        const errorText = await response.text();
        throw new Error(`Error creating checkout session: ${response.status} ${response.statusText} - ${errorText}`);
      }

      const responseData = await response.json();
      const checkoutUrl = responseData.body?.url || responseData.url;

      if (!checkoutUrl) {
        throw new Error('Checkout URL not found in response');
      }

      // Redirigir a la página de checkout de Stripe
      window.location.href = checkoutUrl;

    } catch (error) {
      console.error(''L Error creating checkout session:', error);
      throw error;
    }
  }

  onGoBackToSignup(): void {
    this.showPlanSelection = false;
  }

  // Método para cerrar el pop-up de error de Stripe
  closeStripeError(): void {
    this.showStripeError = false;
    this.stripeErrorMessage = '';
    this.showPlanSelection = true; // Volver a mostrar la selección de planes
  }

  private handleRegistrationError(error: any): void {
    console.log('Registration error:', error);
    // Los errores comunes serán manejados por Firebase
  }
}
```

Ø=ÜÁ features\auth\signup\components\plan-selection

Ø=ÜÄ features\auth\signup\components\plan-selection\plan-selection.component.ts

```
import { Component, EventEmitter, Input, Output } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Router } from '@angular/router';

export interface PlanCard {
  name: string;
  price: number;
  period: string;
```

```
  mostPopular?: boolean;
  icon: 'triangle' | 'circle' | 'square';
  color: string;
  features: {
    label: string;
    value: string | number;
  }[];
  cta: string;
}

@Component({
  selector: 'app-plan-selection',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './plan-selection.component.html',
  styleUrl: './plan-selection.component.scss'
})
export class PlanSelectionComponent {
  @Input() userData: any = null;
  @Output() planSelected = new EventEmitter<PlanCard>();
  @Output() goBack = new EventEmitter<void>();

  plansData: PlanCard[] = [
    {
      name: 'Free',
      price: 0,
      period: '/month',
      icon: 'triangle',
      color: '#4b7ee8',
      features: [
        { label: 'Trading Accounts', value: 1 },
        { label: 'Strategies', value: 1 },
        { label: 'Consistency Rules', value: 'YES' },
        { label: 'Trading Journal', value: 'YES' },
        { label: 'Live Statistics', value: 'YES' }
      ],
      cta: 'Get Free Now'
    },
    {
      name: 'Starter',
      price: 35,
      period: '/month',
      icon: 'circle',
      color: '#4b7ee8',
      features: [
        { label: 'Trading Accounts', value: 2 },
        { label: 'Strategies', value: 3 },
        { label: 'Consistency Rules', value: 'YES' },
        { label: 'Trading Journal', value: 'YES' },
        { label: 'Live Statistics', value: 'YES' }
      ],
      cta: 'Get Starter Now'
    },
    {
      name: 'Pro',
      price: 99,
      period: '/month',
      mostPopular: true,
      icon: 'square',
      color: '#d1ff81',
      features: [
        { label: 'Trading Accounts', value: 6 },
        { label: 'Strategies', value: 8 },
        { label: 'Consistency Rules', value: 'YES' },
        { label: 'Trading Journal', value: 'YES' },
        { label: 'Live Statistics', value: 'YES' }
      ],
      cta: 'Get Pro Now'
    }
  ];
```

```
79
80    constructor(private router: Router) {}
81
82    onPlanSelect(plan: PlanCard): void {
83      this.planSelected.emit(plan);
84    }
85
86    onGoBack(): void {
87      this.goBack.emit();
88    }
89
90    isNumeric(value: string | number): boolean {
91      if (typeof value === 'number') {
92        return true;
93      }
94      if (typeof value === 'string') {
95        return !isNaN(Number(value)) && !isNaN(parseFloat(value));
96      }
97      return false;
98    }
99  }
100
```

Ø=ÜÁ features\auth\store

Ø=ÜÄ features\auth\store\user.actions.ts

```
1  import { createAction, props } from '@ngrx/store';
2  import { User } from '../../overview/models/overview';
3
4  export const setUserData = createAction(
5    '[Auth] Set user data',
6    props<{ user: User | null }>()
7  );
8
```

Ø=ÜÄ features\auth\store\user.reducer.ts

```
1  import { createReducer, on } from '@ngrx/store';
2  import { setUserData } from './user.actions';
3  import { User } from '../../overview/models/overview';
4
5  export interface UserState {
6    user: User | null;
7  }
8
9  export const initialState: UserState = {
10   user: null,
11 };
12
13 export const userReducer = createReducer(
14   initialState,
15   on(setUserData, (state, { user }) => ({ ...state, user }))
16 );
17
```

Ø=ÜÄ features\auth\store\user.selectios.ts

```ts
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { UserState } from './user.reducer';
import e from 'express';

export const selectUser = createFeatureSelector<UserState>('user');

export const getUserId = createSelector(selectUser, (state) => state.user?.id);

export const getUserEmail = createSelector(
  selectUser,
  (state) => state.user?.email
);

export const getUserName = createSelector(
  selectUser,
  (state) => state.user?.firstName
);

export const getUserLastName = createSelector(
  selectUser,
  (state) => state.user?.lastName
);

export const getUserPhoneNumber = createSelector(
  selectUser,
  (state) => state.user?.phoneNumber
);

export const getUserBirthday = createSelector(
  selectUser,
  (state) => state.user?.birthday
);

export const getUserTokenId = createSelector(
  selectUser,
  (state) => state.user?.tokenId
);

export const getIsAdmin = createSelector(
  selectUser,
  (state) => state.user?.isAdmin
);

```

Ø=ÜÁ features\overview

Ø=ÜÄ features\overview\overview.component.ts

```ts
import { Store } from '@ngrx/store';
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { statCardComponent } from '../report/components/statCard/stat_card.component';
import { OverviewService } from './services/overview.service';
import { overviewSubscriptionData, User } from './models/overview';
import { LoadingPopupComponent } from '../../shared/pop-ups/loading-pop-up/loading-
popup.component';
import { FormsModule } from '@angular/forms';
import { TradeSwitchTableComponent } from './components/tradeSwitch-table/
tradeSwitchTable.component';
import { TopListComponent } from './components/top-list/top-list.component';
import { RouterLink } from '@angular/router';
```

```typescript
import { AppContextService } from '../../shared/context';
import { PlanService } from '../../shared/services/planService';
import { SubscriptionService } from '../../shared/services/subscription-service';

/**
 * Main overview component for displaying dashboard statistics and user data.
 *
 * This component provides a comprehensive dashboard view that includes:
 * - User statistics and metrics
 * - Revenue calculations based on subscriptions
 * - Top 10 users by profit
 * - Subscription data overview
 * - CSV export functionality with date range selection
 *
 * Related to:
 * - OverviewService: Fetches user and subscription data
 * - AppContextService: Manages global overview data state
 * - PlanService: Gets plan information for revenue calculation
 * - SubscriptionService: Gets user subscription data
 * - TradeSwitchTableComponent: Displays user table with filtering
 * - TopListComponent: Displays top users list
 *
 * @component
 * @selector app-overview
 * @standalone true
 */
@Component({
  selector: 'app-overview',
  imports: [
    CommonModule,
    statCardComponent,
    LoadingPopupComponent,
    FormsModule,
    TradeSwitchTableComponent,
    TopListComponent,
    RouterLink,
  ],
  templateUrl: './overview.component.html',
  styleUrl: './overview.component.scss',
  standalone: true,
})
export class Overview {
  /** Top 10 users sorted by profit */
  topUsers: User[] = [];

  /**
   * Constructor for Overview component.
   *
   * @param store - NgRx Store (injected but not currently used)
   * @param overviewSvc - Service for fetching overview data
   * @param appContext - Application context service for global state management
   * @param planService - Service for fetching plan information
   * @param subscriptionService - Service for fetching subscription data
   */
  constructor(
    private store: Store,
    private overviewSvc: OverviewService,
    private appContext: AppContextService,
    private planService: PlanService,
    private subscriptionService: SubscriptionService
  ) {}

  loading = false;
  subscriptionsData: overviewSubscriptionData | null = null;
  usersData: User[] = [];
  newUsers = 0;
  newUsersGrowthPercentage = 0;
  calculatedRevenue = 0;
  paidSubscriptions = 0;

```

```
  // Loading granular por sección
  private loadingStates = {
    users: false,
    cards: false,
    revenue: false,
    subscriptions: false,
  };

  // Export modal state
  showExportModal = false;
  exportStartDate: string = '';
  exportEndDate: string = '';
  exportError: string = '';
  showDateDropdown = false;
  // Calendar state (single-picker for range)
  calYear = 0;
  calMonth = 0; // 0-11
  weeks: { date: Date; inMonth: boolean }[][] = [];

  /**
   * Builds a calendar grid for the export date picker.
   *
   * Creates a 6-week grid (42 days) starting from the Sunday of the week
   * containing the first day of the specified month. Each day is marked
   * with whether it belongs to the current month.
   *
   * @private
   * @param year - Year for the calendar
   * @param month - Month for the calendar (0-11, where 0 is January)
   * @memberof Overview
   */
  private buildCalendar(year: number, month: number) {
    // Start from Sunday of the week containing the 1st of the month
    const firstOfMonth = new Date(year, month, 1);
    const start = new Date(firstOfMonth);
    const day = start.getDay(); // 0 Sun .. 6 Sat
    start.setDate(start.getDate() - day);

    const grid: { date: Date; inMonth: boolean }[] = [];
    for (let i = 0; i < 42; i++) {
      const d = new Date(start);
      d.setDate(start.getDate() + i);
      grid.push({ date: d, inMonth: d.getMonth() === month });
    }
    // chunk into weeks
    this.weeks = [];
    for (let i = 0; i < 6; i++) {
      this.weeks.push(grid.slice(i * 7, i * 7 + 7));
    }
  }

  /**
   * Gets the formatted month and year label for the calendar.
   *
   * @returns Formatted string like "January 2024"
   * @memberof Overview
   */
  get monthLabel(): string {
    const m = new Date(this.calYear, this.calMonth, 1);
    return m.toLocaleString(undefined, { month: 'long', year: 'numeric' });
  }

  /**
   * Opens the export modal and initializes the calendar to current month.
   *
   * Resets export error and builds the calendar for the current date.
   *
   * @memberof Overview
   */
  openExportModal() {
```

```
    this.showExportModal = true;
    this.exportError = '';
    const today = new Date();
    this.calYear = today.getFullYear();
    this.calMonth = today.getMonth();
    this.buildCalendar(this.calYear, this.calMonth);
  }

  /**
   * Initializes the component on load.
   *
   * Subscribes to context data and loads all configuration data
   * including users, revenue, and subscription information.
   *
   * @memberof Overview
   */
  ngOnInit(): void {
    this.subscribeToContextData();
    this.loadConfig();
  }

  /**
   * Subscribes to overview data from the application context.
   *
   * Listens to changes in overview data (users and subscriptions)
   * and updates local component state when data changes.
   *
   * Related to:
   * - AppContextService.overviewData$: Observable of overview data
   *
   * @private
   * @memberof Overview
   */
  private subscribeToContextData() {
    // Suscribirse a los datos de overview desde el contexto
    this.appContext.overviewData$.subscribe(data => {
      this.usersData = data.allUsers;
      // Convertir subscriptions a overviewSubscriptionData si es necesario
      this.subscriptionsData = data.subscriptions as any;
    });

    // No usamos el loading global del contexto aquí; control fino local
  }

  /**
   * Loads all configuration data for the overview dashboard.
   *
   * Performs the following operations in sequence:
   * 1. Resets all loading states
   * 2. Loads user data
   * 3. Calculates revenue
   * 4. Loads subscription overview data
   * 5. Checks if all data is loaded to hide loading indicator
   *
   * Related to:
   * - getUsersData(): Fetches and processes user data
   * - calculateRevenue(): Calculates total revenue from subscriptions
   * - getOverviewSubscriptionData(): Fetches subscription statistics
   * - checkAllLoaded(): Verifies all data is loaded
   *
   * @async
   * @memberof Overview
   */
  async loadConfig() {
    this.loading = true;
    this.loadingStates = { users: false, cards: false, revenue: false, subscriptions:
false, };
    await this.getUsersData();
    await this.calculateRevenue();
    this.getOverviewSubscriptionData();
    this.checkAllLoaded();
```

```
  }

  /**
   * Fetches user data from Firebase and processes it.
   *
   * Retrieves all users, filters out admin users, calculates new users
   * for today, and filters top 10 users by profit. Updates loading
   * states when complete.
   *
   * Related to:
   * - OverviewService.getUsersData(): Fetches users from Firebase
   * - calculateNewUsers(): Calculates new users registered today
   * - filterTop10Users(): Filters and sorts top users by profit
   *
   * @async
   * @memberof Overview
   */
  async getUsersData() {
    return this.overviewSvc
      .getUsersData()
      .then((docSnap) => {
        if (docSnap && !docSnap.empty && docSnap.docs.length > 0) {
          this.usersData = docSnap.docs
            .map((doc) => doc.data() as User)
            .filter((user) => !user.isAdmin);

          // Calcular nuevos usuarios basándose en la fecha actual
          this.calculateNewUsers();
          this.filterTop10Users();
          this.loadingStates.users = true;
          this.loadingStates.cards = true; // top users y métricas listas
          this.checkAllLoaded();
        } else {
          this.loadingStates.users = true;
          this.loadingStates.cards = true;
          this.checkAllLoaded();
          console.warn('No config');
        }
      })
      .catch((err) => {
        this.loadingStates.users = true;
        this.loadingStates.cards = true;
        this.checkAllLoaded();
      });
  }

  /**
   * Fetches overview subscription data from Firebase.
   *
   * Retrieves subscription statistics including monthly revenue
   * and user counts. Updates loading state when complete.
   *
   * Related to:
   * - OverviewService.getOverviewSubscriptionData(): Fetches subscription data
   *
   * @memberof Overview
   */
  getOverviewSubscriptionData() {
    this.overviewSvc
      .getOverviewSubscriptionData()
      .then((docSnap) => {
        if (docSnap && !docSnap.empty && docSnap.docs.length > 0) {
          const data = docSnap.docs[0].data() as overviewSubscriptionData;
          this.subscriptionsData = data;
          this.loadingStates.subscriptions = true;
          this.checkAllLoaded();
        } else {
          console.warn('No config');
          this.loadingStates.subscriptions = true;
          this.checkAllLoaded();
```

```
        }
      })
      .catch((err) => {
        this.loadingStates.subscriptions = true;
        this.checkAllLoaded();

        console.error('Error to get the config', err);
      });
  }

  /**
   * Calculates the number of new users registered today.
   *
   * Filters users by subscription_date to find users registered
   * between start and end of current day. Then calculates growth
   * percentage.
   *
   * Related to:
   * - calculateGrowthPercentage(): Calculates percentage of new users
   *
   * @memberof Overview
   */
  calculateNewUsers() {
    const today = new Date();
    const startOfDay = new Date(today.getFullYear(), today.getMonth(), today.getDate());
    const endOfDay = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 1);

    // Convertir las fechas a timestamps para comparar con subscription_date
    const startOfDayTimestamp = startOfDay.getTime();
    const endOfDayTimestamp = endOfDay.getTime();

    // Filtrar usuarios que se registraron hoy
    this.newUsers = this.usersData.filter(user =>
      user.subscription_date >= startOfDayTimestamp &&
      user.subscription_date < endOfDayTimestamp
    ).length;

    // Calcular el porcentaje de crecimiento
    this.calculateGrowthPercentage();
  }

  /**
   * Calculates the growth percentage of new users.
   *
   * Computes the percentage of new users relative to total users.
   * Rounds to 1 decimal place. Returns 0 if there are no users.
   *
   * Formula: (newUsers / totalUsers) * 100
   *
   * @memberof Overview
   */
  calculateGrowthPercentage() {
    const totalUsers = this.usersData.length;

    if (totalUsers === 0) {
      this.newUsersGrowthPercentage = 0;
      return;
    }

    // Calcular el porcentaje de nuevos usuarios respecto al total
    // (nuevos usuarios / total usuarios) * 100
    this.newUsersGrowthPercentage = (this.newUsers / totalUsers) * 100;

    // Redondear a 1 decimal
    this.newUsersGrowthPercentage = Math.round(this.newUsersGrowthPercentage * 10) / 10;
  }

  /**
   * Calculates total revenue from user subscriptions.
   *
```

```
    * This method:
    * 1. Loads all available plans
    * 2. Counts users per plan by checking their latest subscription
    * 3. Calculates revenue for each plan (userCount * planPrice)
    * 4. Sums total revenue across all plans
    * 5. Counts users with paid subscriptions (plans with price > 0)
    *
    * Related to:
    * - PlanService.getAllPlans(): Gets all subscription plans
    * - SubscriptionService.getUserLatestSubscription(): Gets user's current plan
    *
    * @async
    * @memberof Overview
    */
   async calculateRevenue() {
     try {
       // Cargar todos los planes
       const plans = await this.planService.getAllPlans();

       if (plans.length === 0) {
         this.calculatedRevenue = 0;
         this.loadingStates.revenue = true;
         this.checkAllLoaded();
         return;
       }

       // Mapa para contar usuarios por plan
       const planUserCountMap: { [planId: string]: number } = {};

       // Inicializar contadores para cada plan
       plans.forEach(plan => {
         planUserCountMap[plan.id] = 0;
       });

       // Recorrer todos los usuarios y obtener sus subscriptions
       for (const user of this.usersData) {
         try {
           const subscription = await
this.subscriptionService.getUserLatestSubscription(user.id);
           if (subscription && subscription.planId) {
             if (planUserCountMap.hasOwnProperty(subscription.planId)) {
               planUserCountMap[subscription.planId]++;
             }
           }
         } catch (error) {
           console.error(`Error obteniendo subscription para usuario ${user.id}:`, error);
         }
       }

       // Calcular el revenue total y contar usuarios con suscripciones pagas
       let totalRevenue = 0;
       let paidUsersCount = 0;

       for (const plan of plans) {
         const userCount = planUserCountMap[plan.id] || 0;
         const planPrice = parseFloat(plan.price) || 0;
         const revenueForPlan = userCount * planPrice;
         totalRevenue += revenueForPlan;

         // Si el plan tiene precio > 0, contar esos usuarios como suscripciones pagas
         if (planPrice > 0 && userCount > 0) {
           paidUsersCount += userCount;
         }
       }

       this.calculatedRevenue = totalRevenue;
       this.paidSubscriptions = paidUsersCount;
       this.loadingStates.revenue = true;
       this.checkAllLoaded();
     } catch (error) {
```

```
      console.error('Error calculando revenue:', error);
      this.calculatedRevenue = 0;
      this.loadingStates.revenue = true;
      this.checkAllLoaded();
    }
  }

  /**
   * Filters and sorts users to get top 10 by profit.
   *
   * Filters users with profit > 0, sorts them in descending order
   * by profit, and takes the first 10 users.
   *
   * @memberof Overview
   */
  filterTop10Users() {
    this.topUsers = this.usersData
      .filter((user) => user.profit > 0)
      .sort((a, b) => b.profit - a.profit)
      .slice(0, 10);
  }

  /**
   * Checks if all data sections have finished loading.
   *
   * Verifies that users, cards, revenue, and subscriptions data
   * are all loaded. If all are loaded, hides the main loading indicator.
   *
   * @private
   * @memberof Overview
   */
  private checkAllLoaded() {
    const allLoaded = this.loadingStates.users && this.loadingStates.cards &&
this.loadingStates.revenue && this.loadingStates.subscriptions;
    if (allLoaded) {
      this.loading = false;
    }
  }

  // ===== Export Data by Date =====

  /**
   * Closes the export modal and resets all export-related state.
   *
   * Clears export dates, errors, and hides the date dropdown.
   *
   * @memberof Overview
   */
  closeExportModal() {
    this.showExportModal = false;
    this.exportStartDate = '';
    this.exportEndDate = '';
    this.exportError = '';
    this.showDateDropdown = false;
  }

  /**
   * Handles date change in export modal.
   *
   * Resets end date if start date is not selected yet (UX requirement).
   * Clears any export errors.
   *
   * @memberof Overview
   */
  onExportDateChange() {
    // No hard validation for end-only; UX requires start first
    if (!this.exportStartDate && this.exportEndDate) {
      // reset end if start not picked yet
      this.exportEndDate = '';
    }
    this.exportError = '';
```

```
  }

  /**
   * Navigates to the previous month in the calendar.
   *
   * Updates calendar year and month, then rebuilds the calendar grid.
   *
   * @memberof Overview
   */
  prevMonth() {
    const d = new Date(this.calYear, this.calMonth - 1, 1);
    this.calYear = d.getFullYear();
    this.calMonth = d.getMonth();
    this.buildCalendar(this.calYear, this.calMonth);
  }

  /**
   * Navigates to the next month in the calendar.
   *
   * Updates calendar year and month, then rebuilds the calendar grid.
   *
   * @memberof Overview
   */
  nextMonth() {
    const d = new Date(this.calYear, this.calMonth + 1, 1);
    this.calYear = d.getFullYear();
    this.calMonth = d.getMonth();
    this.buildCalendar(this.calYear, this.calMonth);
  }

  /**
   * Handles day selection in the calendar for date range picker.
   *
   * Implements a two-click date range selection:
   * - First click: Sets start date
   * - Second click: Sets end date (if after start) or moves start (if before)
   * - Third click: Starts new selection
   *
   * @param day - Selected date from calendar
   * @memberof Overview
   */
  onDayPick(day: Date) {
    const iso = (d: Date) => this.formatLocalDate(d);
    if (!this.exportStartDate) {
      this.exportStartDate = iso(day);
      this.exportEndDate = '';
      return;
    }
    const start = this.parseLocalDate(this.exportStartDate);
    if (!this.exportEndDate) {
      if (day < start) {
        // If clicked before start, move start to that day
        this.exportStartDate = iso(day);
      } else {
        this.exportEndDate = iso(day);
      }
      return;
    }
    // If both set, start a new selection
    this.exportStartDate = iso(day);
    this.exportEndDate = '';
  }

  /**
   * Checks if a calendar day is within the selected date range.
   *
   * Returns true if the day falls between start and end dates
   * (inclusive). Handles cases where only start date is selected.
   *
   * @param day - Date to check
```

```
    * @returns true if day is in selected range, false otherwise
    * @memberof Overview
    */
   isSelected(day: Date): boolean {
     if (!this.exportStartDate && !this.exportEndDate) return false;
     const time = new Date(day.getFullYear(), day.getMonth(), day.getDate()).getTime();
     const start = this.exportStartDate
       ? this.parseLocalDate(this.exportStartDate).setHours(0, 0, 0, 0)
       : Number.NaN;
     const end = this.exportEndDate
       ? this.parseLocalDate(this.exportEndDate).setHours(23, 59, 59, 999)
       : start;
     return time >= start && time <= end;
   }

   /**
    * Exports user data to CSV file with optional date filtering.
    *
    * This method:
    * 1. Determines date range from selected dates (or exports all if no dates)
    * 2. Filters users by subscription_date within the range
    * 3. Generates CSV content with headers and user data
    * 4. Creates a downloadable blob and triggers download
    * 5. Closes the export modal
    *
    * CSV includes: User ID, Name, Email, Status, Strategies, Trading Accounts,
    * Strategy Followed %, Net PnL, Profit, Best Trade, Subscription Date
    *
    * Related to:
    * - parseLocalDate(): Parses date strings
    * - escapeCsv(): Escapes special characters in CSV values
    * - closeExportModal(): Closes modal after export
    *
    * @memberof Overview
    */
   exportDataAsCSV() {
     // Determine range
     let startTs: number | null = null;
     let endTs: number | null = null;

     if (this.exportStartDate) {
       const start = this.parseLocalDate(this.exportStartDate);
       // start of day
       startTs = new Date(start.getFullYear(), start.getMonth(), start.getDate()).getTime();
       if (this.exportEndDate) {
         const end = this.parseLocalDate(this.exportEndDate);
         // end of day (inclusive)
         endTs = new Date(end.getFullYear(), end.getMonth(), end.getDate(), 23, 59, 59,
999).getTime();
       } else {
         // If only start provided, range is only that day
         endTs = new Date(start.getFullYear(), start.getMonth(), start.getDate(), 23, 59, 59,
999).getTime();
       }
     }

     // Filter users by subscription_date (fallback to lastUpdated if needed)
     const filtered = this.usersData.filter(u => {
       const ts = (u.subscription_date ?? u.lastUpdated) || 0;
       if (startTs !== null && endTs !== null) {
         return ts >= startTs && ts <= endTs;
       }
       return true; // no dates -> export all
     });

     const rows: string[] = [];
     // Header
     rows.push([
       'User ID', 'First Name', 'Last Name', 'Email', 'Status', 'Strategies', 'Trading
Accounts', '% Strat Followed', 'Net PnL', 'Profit', 'Best Trade', 'Subscription Date'
     ].join(','));

     // Data
```

```
    for (const u of filtered) {
      const subDate = u.subscription_date ? new Date(u.subscription_date).toISOString() : '';
      rows.push([
        `${u.id ?? ''}`,
        this.escapeCsv(u.firstName ?? ''),
        this.escapeCsv(u.lastName ?? ''),
        this.escapeCsv(u.email ?? ''),
        `${u.status ?? ''}`,
        `${u.strategies ?? 0}`,
        `${u.trading_accounts ?? 0}`,
        `${u.strategy_followed ?? 0}`,
        `${u.netPnl ?? 0}`,
        `${u.profit ?? 0}`,
        `${u.best_trade ?? 0}`,
        subDate,
      ].join(','));
    }

    const csvContent = rows.join('\n');
    const blob = new Blob([csvContent], { type: 'text/csv;charset=utf-8;' });
    const url = URL.createObjectURL(blob);
    const link = document.createElement('a');
    link.href = url;
    const filename = this.exportStartDate || this.exportEndDate ?
`export_${Date.now()}.csv` : 'export_all.csv';
    link.setAttribute('download', filename);
    link.click();
    URL.revokeObjectURL(url);

    this.closeExportModal();
  }

  /**
   * Escapes special characters in CSV values.
   *
   * Wraps value in quotes if it contains comma, quote, or newline.
   * Doubles any quotes within the value.
   *
   * @private
   * @param value - String value to escape
   * @returns Escaped CSV value
   * @memberof Overview
   */
  private escapeCsv(value: string): string {
    if (value.includes(',') || value.includes('"') || value.includes('\n')) {
      return '"' + value.replace(/\"/g, '\"\"') + '"';
    }
    return value;
  }

  /**
   * Formats a Date object to YYYY-MM-DD string format.
   *
   * @private
   * @param d - Date to format
   * @returns Formatted date string (YYYY-MM-DD)
   * @memberof Overview
   */
  private formatLocalDate(d: Date): string {
    const y = d.getFullYear();
    const m = String(d.getMonth() + 1).padStart(2, '0');
    const day = String(d.getDate()).padStart(2, '0');
    return `${y}-${m}-${day}`;
  }

  /**
   * Parses a YYYY-MM-DD string to a Date object.
   *
   * @private
   * @param s - Date string in YYYY-MM-DD format
   * @returns Parsed Date object
```

```
    * @memberof Overview
    */
   private parseLocalDate(s: string): Date {
     // Expecting YYYY-MM-DD
     const [y, m, d] = s.split('-').map(Number);
     return new Date(y, (m || 1) - 1, d || 1);
   }
}

```

Ø=ÜÁ features\overview\components\top-list

Ø=ÜÄ features\overview\components\top-list\top-list.component.ts

```typescript
import { CommonModule } from '@angular/common';
import { Component, Input } from '@angular/core';
import { User, UserStatus } from '../../models/overview';

/**
 * Component for displaying a single user in the top users list.
 *
 * This component displays user information in a card format,
 * showing user initials, name, and profit. It's used in the
 * overview dashboard to show the top 10 users by profit.
 *
 * Related to:
 * - OverviewComponent: Passes user data as Input
 *
 * @component
 * @selector app-top-list
 * @standalone true
 */
@Component({
  selector: 'app-top-list',
  templateUrl: './top-list.component.html',
  styleUrls: ['./top-list.component.scss'],
  standalone: true,
  imports: [CommonModule],
})
export class TopListComponent {
  @Input() user: User = {
    best_trade: 0,
    birthday: new Date(),
    firstName: '',
    id: '',
    lastName: '',
    netPnl: 0,
    number_trades: 0,
    phoneNumber: '',
    profit: 0,
    status: UserStatus.CREATED,
    strategy_followed: 0,
    subscription_date: 0,
    tokenId: '',
    email: '',
    total_spend: 0,
    isAdmin: false,
    lastUpdated: 0,
    trading_accounts: 0,
    strategies: 0,
  };

  constructor() {}
```

```
50
51    /**
52     * Gets user initials from first and last name.
53     *
54     * Takes the first character of firstName and lastName,
55     * converts them to uppercase, and concatenates them.
56     *
57     * @param user - User object containing firstName and lastName
58     * @returns Two-letter initials string (e.g., "JD" for John Doe)
59     * @memberof TopListComponent
60     */
61    onlyNameInitials(user: User) {
62      return (
63        user.firstName.charAt(0).toUpperCase() +
64        user.lastName.charAt(0).toUpperCase()
65      );
66    }
67
68    /**
69     * Formats profit value for display.
70     *
71     * Formats profit with appropriate currency symbol and scaling:
72     * - $0 for zero
73     * - $X.XX for values less than 1000
74     * - $X.XK for values 1000 or greater (e.g., $5.2K for 5200)
75     *
76     * @param profit - Profit value to format
77     * @returns Formatted profit string
78     * @memberof TopListComponent
79     */
80    formatProfit(profit: number): string {
81      if (profit === 0) return '$0';
82      if (Math.abs(profit) < 1000) {
83        return `$${profit.toFixed(2)}`;
84      }
85      const k = profit / 1000;
86      return `$${k.toFixed(1)}K`;
87    }
88  }
89
```

Ø=ÜÁ features\overview\components\tradeSwitch-table

Ø=ÜÄ features\overview\components\tradeSwitch-table\tradeSwitchTable.component.ts

```
1  import { Component, Input, Injectable, HostListener, ElementRef, ViewChild, OnInit,
2  OnDestroy } from '@angular/core';
3  import { CommonModule } from '@angular/common';
4  import { User, UserStatus } from '../../models/overview';
5  import { FormsModule } from '@angular/forms';
6  import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';
7  import { AuthService } from '../../../../features/auth/service/authService';
8  import { AccountData } from '../../../../features/auth/models/userModel';
9
10 /**
11  * Component for displaying a comprehensive user table with filtering and pagination.
12  *
13  * This component provides a detailed table view of all users with the following features:
14  * - Search by user name
15  * - Filter by status, strategy followed percentage, number of strategies, and trading
16 accounts
17  * - Expandable rows showing user's trading accounts
18  * - Pagination for large datasets
19  * - CSV-friendly data display
20  *
```

```
 * Related to:
 * - OverviewComponent: Receives users array as Input
 * - AuthService: Fetches user trading accounts
 * - NumberFormatterService: Formats currency values
 *
 * @component
 * @selector app-trade-switch-table
 * @standalone true
 */
@Component({
  selector: 'app-trade-switch-table',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './tradeSwitchTable.component.html',
  styleUrls: ['./tradeSwitchTable.component.scss'],
})
@Injectable()
export class TradeSwitchTableComponent implements OnInit, OnDestroy {
  @Input() users: User[] = [];
  @ViewChild('filterModal', { static: false }) filterModal!: ElementRef;
  @ViewChild('filterButton', { static: false }) filterButton!: ElementRef;

  // Mapa de userId -> AccountData[]
  userAccountsMap: Map<string, AccountData[]> = new Map();
  // Set de userIds expandidos
  expandedUsers: Set<string> = new Set();
  // Flag para saber si las cuentas están cargadas
  accountsLoaded = false;

  // Valores iniciales (usados en el formulario)
  initialStatus: UserStatus | string = '';
  initialMinStrat: number = 0;
  initialMaxStrat: number = 100;
  initialStrategies: number = 0; // 0-8, si es 0 no filtra
  initialTradingAccounts: number = 0; // 0-8, si es 0 no filtra

  // Valores aplicados (usados para filtrar)
  appliedStatus: UserStatus | string = '';
  appliedMinStrat: number = 0;
  appliedMaxStrat: number = 100;
  appliedStrategies: number = 0; // 0-8, si es 0 no filtra
  appliedTradingAccounts: number = 0; // 0-8, si es 0 no filtra

  showFilter = false;
  currentPage: number = 1;
  itemsPerPage: number = 10;

  private numberFormatter = new NumberFormatterService();

  constructor(private authService: AuthService) {}

  ngOnInit() {
    this.loadAllAccounts();
  }

  ngOnDestroy() {
    // Cleanup si es necesario
  }

  /**
   * Loads all trading accounts and groups them by userId.
   *
   * Fetches all accounts from Firebase and creates a map
   * where each userId maps to an array of their accounts.
   *
   * Related to:
   * - AuthService.getAllAccounts(): Fetches all accounts from Firebase
   *
   * @async
   * @memberof TradeSwitchTableComponent
```

```
   */
  async loadAllAccounts() {
    try {
      const allAccounts = await this.authService.getAllAccounts();
      if (allAccounts) {
        // Agrupar cuentas por userId
        const accountsMap = new Map<string, AccountData[]>();
        allAccounts.forEach(account => {
          if (!accountsMap.has(account.userId)) {
            accountsMap.set(account.userId, []);
          }
          accountsMap.get(account.userId)!.push(account);
        });
        this.userAccountsMap = accountsMap;
      }
      this.accountsLoaded = true;
    } catch (error) {
      console.error('Error loading accounts:', error);
      this.accountsLoaded = true;
    }
  }

  /**
   * Gets trading accounts for a specific user.
   *
   * @param userId - User ID to get accounts for
   * @returns Array of AccountData for the user, or empty array if none
   * @memberof TradeSwitchTableComponent
   */
  getUserAccounts(userId: string): AccountData[] {
    return this.userAccountsMap.get(userId) || [];
  }

  /**
   * Checks if a user has any trading accounts.
   *
   * @param userId - User ID to check
   * @returns true if user has accounts, false otherwise
   * @memberof TradeSwitchTableComponent
   */
  hasAccounts(userId: string): boolean {
    const accounts = this.userAccountsMap.get(userId);
    return accounts ? accounts.length > 0 : false;
  }

  /**
   * Toggles the expansion state of a user row.
   *
   * Adds or removes the userId from the expandedUsers set
   * to show/hide the user's trading accounts.
   *
   * @param userId - User ID to toggle expansion for
   * @memberof TradeSwitchTableComponent
   */
  toggleExpand(userId: string) {
    if (this.expandedUsers.has(userId)) {
      this.expandedUsers.delete(userId);
    } else {
      this.expandedUsers.add(userId);
    }
  }

  /**
   * Checks if a user row is currently expanded.
   *
   * @param userId - User ID to check
   * @returns true if user row is expanded, false otherwise
   * @memberof TradeSwitchTableComponent
   */
  isExpanded(userId: string): boolean {
```

```
    return this.expandedUsers.has(userId);
  }

  private _searchTerm = '';

  /**
   * Gets the current search term.
   *
   * @returns Current search term string
   * @memberof TradeSwitchTableComponent
   */
  get searchTerm(): string {
    return this._searchTerm;
  }

  /**
   * Sets the search term and resets to first page.
   *
   * When search term changes, automatically navigates to page 1
   * to show filtered results from the beginning.
   *
   * @param val - New search term value
   * @memberof TradeSwitchTableComponent
   */
  set searchTerm(val: string) {
    this._searchTerm = val;
    this.goToPage(1);
  }

  /**
   * Host listener for clicks outside the filter modal.
   *
   * Closes the filter modal when user clicks outside of it
   * or the filter button. Prevents closing when clicking inside.
   *
   * @param event - Mouse click event
   * @memberof TradeSwitchTableComponent
   */
  @HostListener('document:click', ['$event'])
  onClickOutside(event: MouseEvent) {
    if (this.showFilter &&
        this.filterModal?.nativeElement &&
        this.filterButton?.nativeElement &&
        !this.filterModal.nativeElement.contains(event.target) &&
        !this.filterButton.nativeElement.contains(event.target)) {
      this.closeFilter();
    }
  }

  /**
   * Getter that returns filtered users based on search term and applied filters.
   *
   * Filters users by:
   * - Search term: Matches against first name and last name
   * - Status: Matches user's display status
   * - Strategy followed: Range between min and max percentage
   * - Strategies: Exact number match (0 means no filter)
   * - Trading accounts: Exact number match (0 means no filter)
   *
   * @returns Array of filtered User objects
   * @memberof TradeSwitchTableComponent
   */
  get filteredUsers(): User[] {
    const lower = this._searchTerm.trim().toLowerCase();

    return this.users.filter((user) => {
      const matchesSearch = `${user.firstName.split(' ')[0]} ${
        user.lastName.split(' ')[0]
      }`
        .toLowerCase()
```

```
        .includes(lower);

      const userDisplayStatus = this.statusClass(user);
      const matchesStatus =
        !this.appliedStatus || this.appliedStatus === '' || userDisplayStatus ===
this.appliedStatus;
      let matchesMinStrat = user.strategy_followed !== undefined && user.strategy_followed
>= (this.appliedMinStrat ?? 0);
      let matchesMaxStrat = user.strategy_followed !== undefined && user.strategy_followed
<= (this.appliedMaxStrat ?? 100);
      if (user.strategy_followed === undefined) {
        matchesMinStrat = false;
        matchesMaxStrat = false;
      }

      // Si appliedStrategies es 0, no se filtra; si es > 0, se filtra por ese número exacto
      const matchesStrategies = this.appliedStrategies === 0 || (user.strategies ?? 0) ===
this.appliedStrategies;
      // Si appliedTradingAccounts es 0, no se filtra; si es > 0, se filtra por ese número
exactoconst matchesTradingAccounts = this.appliedTradingAccounts === 0 ||
(user.trading_accounts ?? 0) === this.appliedTradingAccounts;
      return (
        matchesSearch &&
        matchesStatus &&
        matchesMinStrat &&
        matchesMaxStrat &&
        matchesStrategies &&
        matchesTradingAccounts
      );
    });
  }

  /**
   * Getter that returns paginated users for current page.
   *
   * Calculates the slice of filtered users to display based on
   * currentPage and itemsPerPage.
   *
   * @returns Array of users for current page
   * @memberof TradeSwitchTableComponent
   */
  get paginatedUsers(): User[] {
    const start = (this.currentPage - 1) * this.itemsPerPage;
    const end = start + this.itemsPerPage;
    return this.filteredUsers.slice(start, end);
  }

  /**
   * Getter that calculates total number of pages.
   *
   * Based on filtered users count and items per page.
   *
   * @returns Total number of pages
   * @memberof TradeSwitchTableComponent
   */
  get totalPages(): number {
    return Math.ceil(this.filteredUsers.length / this.itemsPerPage);
  }

  /**
   * Determines the display status class for a user.
   *
   * Status logic:
   * - "banned" if user status is banned
   * - "created" if all user metrics are zero (new/inactive user)
   * - "active" if user has any activity (non-zero metrics)
   *
   * @param user - User object to determine status for
   * @returns Status class string ('banned', 'created', or 'active')
   * @memberof TradeSwitchTableComponent
   */
```

```
  statusClass(user: User): string {
    // Si el status es banned, retornar banned
    if (String(user.status) === 'banned') {
      return 'banned';
    }

    // Verificar si todos los valores están en 0
    const allValuesZero =
      (user.trading_accounts ?? 0) === 0 &&
      (user.strategies ?? 0) === 0 &&
      (user.strategy_followed ?? 0) === 0 &&
      (user.netPnl ?? 0) === 0 &&
      (user.profit ?? 0) === 0 &&
      (user.number_trades ?? 0) === 0 &&
      (user.total_spend ?? 0) === 0;

    // Si todos los valores están en 0, retornar created
    if (allValuesZero) {
      return 'created';
    }

    // Si no todos están en 0, retornar active
    return 'active';
  }

  /**
   * Gets display status string with capitalized first letter.
   *
   * @param user - User object to get status for
   * @returns Capitalized status string (e.g., "Active", "Created", "Banned")
   * @memberof TradeSwitchTableComponent
   */
  getDisplayStatus(user: User): string {
    const status = this.statusClass(user);
    return status.charAt(0).toUpperCase() + status.slice(1);
  }

  /**
   * Determines CSS class for return value display.
   *
   * Returns 'green' for positive or zero values, 'red' for negative values.
   *
   * @param returnValue - Numeric value to determine class for
   * @returns CSS class string ('green' or 'red')
   * @memberof TradeSwitchTableComponent
   */
  returnClass(returnValue: number) {
    return returnValue >= 0 ? 'green' : 'red';
  }

  /**
   * Toggles the filter panel visibility.
   *
   * @memberof TradeSwitchTableComponent
   */
  openFilter() {
    this.showFilter = !this.showFilter;
  }

  /**
   * Closes the filter panel.
   *
   * @memberof TradeSwitchTableComponent
   */
  closeFilter() {
    this.showFilter = false;
  }

  /**
   * Applies filter values from initial to applied state.
```

```
   *
   * Copies all initial filter values to applied values,
   * closes the filter panel, and applies the filters.
   *
   * Related to:
   * - applyFilters(): Applies the filters and resets pagination
   *
   * @memberof TradeSwitchTableComponent
   */
  apply() {
    // Aplicar los valores iniciales a los aplicados
    this.appliedStatus = this.initialStatus;
    this.appliedMinStrat = this.initialMinStrat;
    this.appliedMaxStrat = this.initialMaxStrat;
    this.appliedStrategies = this.initialStrategies;
    this.appliedTradingAccounts = this.initialTradingAccounts;

    this.showFilter = false;
    this.applyFilters();
  }

  /**
   * Applies filters and resets to first page.
   *
   * Navigates to page 1 when filters are applied to show
   * filtered results from the beginning.
   *
   * @memberof TradeSwitchTableComponent
   */
  applyFilters() {
    this.goToPage(1);
  }

  /**
   * Resets all filters to default values.
   *
   * Clears both initial and applied filter values,
   * closes the filter panel, and applies the reset filters.
   *
   * @memberof TradeSwitchTableComponent
   */
  resetFilters() {
    this.initialStatus = '';
    this.initialMinStrat = 0;
    this.initialMaxStrat = 100;
    this.initialStrategies = 0;
    this.initialTradingAccounts = 0;

    this.appliedStatus = '';
    this.appliedMinStrat = 0;
    this.appliedMaxStrat = 100;
    this.appliedStrategies = 0;
    this.appliedTradingAccounts = 0;

    this.showFilter = false; // Cerrar el filter
    this.applyFilters();
  }

  /**
   * Getter that returns the strategy followed percentage range as a formatted string.
   *
   * Ensures min is not greater than max by using Math.min/max.
   * Used for real-time display in the filter UI.
   *
   * @returns Formatted range string (e.g., "0% - 100%")
   * @memberof TradeSwitchTableComponent
   */
  get stratFollowedRange(): string {
    // Asegurar que min no sea mayor que max
    const min = Math.min(this.initialMinStrat, this.initialMaxStrat);
```

```
    const max = Math.max(this.initialMinStrat, this.initialMaxStrat);
    return `${min}% - ${max}%`;
  }

  /**
   * Ensures minStrat is not greater than maxStrat.
   *
   * If minStrat exceeds maxStrat, swaps the values.
   * Called when minStrat input changes.
   *
   * @memberof TradeSwitchTableComponent
   */
  onMinStratChange() {
    if (this.initialMinStrat > this.initialMaxStrat) {
      const temp = this.initialMinStrat;
      this.initialMinStrat = this.initialMaxStrat;
      this.initialMaxStrat = temp;
    }
  }

  /**
   * Ensures maxStrat is not less than minStrat.
   *
   * If maxStrat is less than minStrat, swaps the values.
   * Called when maxStrat input changes.
   *
   * @memberof TradeSwitchTableComponent
   */
  onMaxStratChange() {
    if (this.initialMaxStrat < this.initialMinStrat) {
      const temp = this.initialMaxStrat;
      this.initialMaxStrat = this.initialMinStrat;
      this.initialMinStrat = temp;
    }
  }

  /**
   * Gets user initials from first and last name.
   *
   * Takes first character of firstName and lastName and concatenates them.
   *
   * @param user - User object containing firstName and lastName
   * @returns Two-letter initials string
   * @memberof TradeSwitchTableComponent
   */
  onlyNameInitials(user: User) {
    return user.firstName.charAt(0) + user.lastName.charAt(0);
  }

  /**
   * Navigates to a specific page.
   *
   * Validates page number is within valid range (1 to totalPages)
   * and updates currentPage.
   *
   * @param page - Page number to navigate to
   * @memberof TradeSwitchTableComponent
   */
  goToPage(page: number) {
    if (page < 1) page = 1;
    if (page > this.totalPages) page = this.totalPages;
    this.currentPage = page;
  }

  /**
   * Navigates to the previous page.
   *
   * @memberof TradeSwitchTableComponent
   */
  prevPage() {
```

```
    this.goToPage(this.currentPage - 1);
  }

  /**
   * Navigates to the next page.
   *
   * @memberof TradeSwitchTableComponent
   */
  nextPage() {
    this.goToPage(this.currentPage + 1);
  }

  /**
   * Formats a numeric value as currency.
   *
   * Uses NumberFormatterService to format the value with
   * appropriate currency symbol and formatting.
   *
   * Related to:
   * - NumberFormatterService.formatCurrency(): Formats currency values
   *
   * @param value - Numeric value to format (can be null or undefined)
   * @returns Formatted currency string
   * @memberof TradeSwitchTableComponent
   */
  formatCurrency(value: number | null | undefined): string {
    return this.numberFormatter.formatCurrency(value);
  }
}

```

Ø=ÜÁ features\overview\models

Ø=ÜÄ features\overview\models\overview.ts

```
/**
 * Enumeration of possible user statuses in the system.
 *
 * Represents the different states a user account can have,
 * from creation to active use, cancellation, or administrative actions.
 *
 * @enum UserStatus
 */
export enum UserStatus {
  ADMIN = 'admin',
  CREATED = 'created',
  PURCHASED = 'purchased',
  PENDING = 'pending',
  ACTIVE = 'active',
  PROCESSING = 'processing',
  CANCELLED = 'cancelled',
  EXPIRED = 'expired',
  BANNED = 'banned',
}

/**
 * Interface representing subscription overview statistics.
 *
 * Contains aggregated data about subscriptions for a specific month,
 * including revenue and user count.
 *
 * Used in:
 * - OverviewComponent: Displays subscription statistics
```

```
 * - OverviewService: Fetches subscription data from Firebase
 *
 * @interface overviewSubscriptionData
 */
export interface overviewSubscriptionData {
  month: string;
  revenue: number;
  users: number;
}

/**
 * Interface representing a user in the system.
 *
 * Contains comprehensive user information including personal data,
 * trading statistics, account information, and subscription details.
 *
 * Used throughout the overview module for displaying user data,
 * calculating statistics, and filtering users.
 *
 * @interface User
 */
export interface User {
  id: any;
  best_trade: number;
  birthday: Date;
  email: string;
  firstName: string;
  lastName: string;
  netPnl: number;
  number_trades: number;
  phoneNumber: string;
  profit: number;
  status: UserStatus;
  strategy_followed: number;
  subscription_date: number;
  lastUpdated: number;
  tokenId: string;
  total_spend: number;
  trading_accounts: number;
  strategies: number;
  isAdmin: boolean;
}

```

Ø=ÜÁ features\overview\services

Ø=ÜÄ features\overview\services\overview.service.ts

```
import { Injectable } from '@angular/core';
import { OverviewDataService } from '../../../shared/services/overview-data.service';
import { AppContextService } from '../../../shared/context';

/**
 * Service for fetching and managing overview data.
 *
 * This service acts as a bridge between the OverviewComponent and
 * the data layer, handling loading states and error management
 * through the AppContextService.
 *
 * Related to:
 * - OverviewDataService: Fetches raw data from Firebase
 * - AppContextService: Manages global loading states and data updates
 *
```

```
 * @injectable
 * @providedIn root
 */
@Injectable({ providedIn: 'root' })
export class OverviewService {
  /**
   * Constructor for OverviewService.
   *
   * @param overviewDataService - Service for fetching overview data from Firebase
   * @param appContext - Application context service for state management
   */
  constructor(
    private overviewDataService: OverviewDataService,
    private appContext: AppContextService
  ) {}

  /**
   * Fetches overview subscription data from Firebase.
   *
   * Manages loading state and error handling through AppContextService.
   * Updates the global overview subscriptions data when successful.
   *
   * Related to:
   * - OverviewDataService.getOverviewSubscriptionData(): Fetches data from Firebase
   * - AppContextService.setLoading(): Manages loading state
   * - AppContextService.updateOverviewSubscriptions(): Updates global state
   * - AppContextService.setError(): Manages error state
   *
   * @async
   * @returns Promise resolving to subscription data document snapshot
   * @throws Error if data fetch fails
   */
  async getOverviewSubscriptionData() {
    this.appContext.setLoading('overview', true);
    this.appContext.setError('overview', null);

    try {
      const subscriptions = await this.overviewDataService.getOverviewSubscriptionData();
      this.appContext.updateOverviewSubscriptions(subscriptions?.docs?.map(doc =>
doc.data()) || []);
      this.appContext.setLoading('overview', false);
      return subscriptions;
    } catch (error) {
      this.appContext.setLoading('overview', false);
      this.appContext.setError('overview', 'Error al obtener datos de suscripciones');
      throw error;
    }
  }

  /**
   * Fetches user data from Firebase.
   *
   * Manages loading state and error handling through AppContextService.
   * Updates the global overview users data when successful.
   *
   * Related to:
   * - OverviewDataService.getUsersData(): Fetches data from Firebase
   * - AppContextService.setLoading(): Manages loading state
   * - AppContextService.updateOverviewUsers(): Updates global state
   * - AppContextService.setError(): Manages error state
   *
   * @async
   * @returns Promise resolving to users data document snapshot
   * @throws Error if data fetch fails
   */
  async getUsersData() {
    this.appContext.setLoading('overview', true);
    this.appContext.setError('overview', null);

    try {
      const users = await this.overviewDataService.getUsersData();
```

```
      const usersData = users?.docs?.map(doc => doc.data() as any) || [];
      this.appContext.updateOverviewUsers(usersData);
      this.appContext.setLoading('overview', false);
      return users;
    } catch (error) {
      this.appContext.setLoading('overview', false);
      this.appContext.setError('overview', 'Error al obtener datos de usuarios');
      throw error;
    }
  }
}

```

Ø=ÜÁ features\report

Ø=ÜÄ features\report\report.component.ts

```
import { Component, Inject, OnInit, PLATFORM_ID } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { CommonModule, isPlatformBrowser } from '@angular/common';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { Store } from '@ngrx/store';
import { AppContextService } from '../../shared/context';
import {
  getUserKey,
  setAvgWnL,
  setGroupedTrades,
  setNetPnL,
  setProfitFactor,
  setTotalTrades,
  setTradeWin,
  setUserKey,
} from './store/report.actions';
import { selectGroupedTrades, selectReport } from './store/report.selectors';
import { interval, last, map, Subscription } from 'rxjs';
import { ReportService } from './service/report.service';
import {
  displayConfigData,
  GroupedTrade,
  GroupedTradeFinal,
  MonthlyReport,
  PluginHistoryRecord,
  StatConfig,
} from './models/report.model';
import {
  calculateAvgWinLossTrades,
  calculateNetPnl,
  calculateProfitFactor,
  calculateTotalTrades,
  calculateTradeWinPercent,
} from './utils/normalization-utils';
import { statCardComponent } from './components/statCard/stat_card.component';
import { PnlGraphComponent } from './components/pnlGraph/pnlGraph.component';
import { CalendarComponent } from './components/calendar/calendar.component';
import { SettingsService } from '../strategy/service/strategy.service';
import { resetConfig } from '../strategy/store/strategy.actions';
import { ConfigurationOverview, RuleType, StrategyState } from '../strategy/models/
strategy.models';
import { WinLossChartComponent } from './components/winLossChart/win-loss-chart.component';
import moment from 'moment-timezone';
import { Router } from '@angular/router';
import { User } from '../overview/models/overview';
import { selectUser } from '../auth/store/user.selectios';
```

```typescript
import { AuthService } from '../auth/service/authService';
import { getBestTrade, getTotalSpend } from './utils/firebase-data-utils';
import { Timestamp } from 'firebase/firestore';
import { initialStrategyState } from '../strategy/store/strategy.reducer';
import { AccountData } from '../auth/models/userModel';
import { PlanLimitationsGuard } from '../../guards/plan-limitations.guard';
import { PlanLimitationModalData } from '../../shared/interfaces/plan-limitation-modal.interface';
import { PlanLimitationModalComponent } from '../../shared/components/plan-limitation-modal/plan-limitation-modal.component';
import { StrategyCardData } from '../../shared/components/strategy-card/strategy-card.interface';
import { LoadingSpinnerComponent } from '../../shared/components/loading-spinner/loading-spinner.component';
import { PluginHistoryService, PluginHistory } from '../../shared/services/plugin-history.service';
import { TimezoneService } from '../../shared/services/timezone.service';

/**
 * Main component for displaying trading reports and analytics.
 *
 * This component is the central hub for displaying comprehensive trading data including:
 * - Trading statistics (Net PnL, Win Rate, Profit Factor, etc.)
 * - PnL charts with monthly/yearly views
 * - Calendar view of trades with strategy compliance
 * - Win/Loss ratio visualization
 * - Account balance information
 * - Strategy configuration display
 *
 * Key Features:
 * - Fetches trading history from TradeLocker API
 * - Processes and groups trades by position
 * - Calculates trading statistics
 * - Manages multiple trading accounts
 * - Caches data in localStorage for performance
 * - Updates monthly reports in Firebase
 * - Handles plan limitations and access control
 *
 * Data Flow:
 * 1. Component initializes and loads saved data from localStorage
 * 2. Subscribes to AppContextService for user, accounts, and strategies
 * 3. Fetches fresh data from API for current account
 * 4. Processes trades and calculates statistics
 * 5. Updates NgRx store and AppContextService
 * 6. Displays data in child components (charts, calendar, stats)
 *
 * Relations:
 * - ReportService: Fetches trading data from API
 * - AppContextService: Global state management
 * - Store (NgRx): Local state for report data
 * - AuthService: User authentication and account management
 * - SettingsService: Strategy configuration
 * - CalendarComponent: Calendar view of trades
 * - PnlGraphComponent: PnL chart visualization
 * - WinLossChartComponent: Win/loss ratio chart
 * - statCardComponent: Individual statistic cards
 *
 * @component
 * @selector app-report
 * @standalone true
 */
@Component({
  selector: 'app-report',
  templateUrl: './report.component.html',
  styleUrls: ['./report.component.scss'],
  standalone: true,
  imports: [
    CommonModule,
    FormsModule,
    ReactiveFormsModule,
    statCardComponent,
    PnlGraphComponent,
    CalendarComponent,
    WinLossChartComponent,
    PlanLimitationModalComponent,
```

```
    LoadingSpinnerComponent,
  ],
})
export class ReportComponent implements OnInit {
  accessToken: string | null = null;
  accountDetails: any = null;
  accountsData: AccountData[] = [];
  accountHistory: GroupedTradeFinal[] = [];
  errorMessage: string | null = null;
  stats?: StatConfig;
  userKey!: string;
  config!: displayConfigData[];
  loading = false;
  fromDate = '';
  toDate = '';
  user: User | null = null;
  requestYear: number = 0;
  private updateSubscription?: Subscription;
  private loadingTimeout?: any;
  strategies: ConfigurationOverview[] = [];

  // Account management
  currentAccount: AccountData | null = null;
  showAccountDropdown = false;
  showReloadButton = false;

  // Balance data from API
  balanceData: any = null;

  // Loading state tracking for complete data loading
  private loadingStates = {
    userData: false,
    accounts: false,
    strategies: false,
    userKey: false,
    historyData: false,
    balanceData: false,
    config: false
  };

  // Flag para rastrear si hay peticiones en curso
  private hasPendingRequests = false;

  // Flag para rastrear si las estadísticas están completamente procesadas
  private statsProcessed = false;

  // Flag para rastrear si las gráficas están completamente renderizadas
  private chartsRendered = false;

  // Local storage keys
  private readonly STORAGE_KEYS = {
    REPORT_DATA: 'tradeSwitch_reportData',
    ACCOUNTS_DATA: 'tradeSwitch_accountsData',
    CURRENT_ACCOUNT: 'tradeSwitch_currentAccount',
    USER_DATA: 'tradeSwitch_userData'
  };

  // Plan limitation modal
  planLimitationModal: PlanLimitationModalData = {
    showModal: false,
    modalType: 'blocked',
    title: '',
    message: '',
    primaryButtonText: '',
    onPrimaryAction: () => {}
  };

  constructor(
    @Inject(PLATFORM_ID) private platformId: any,
    private store: Store,
```

```
    private reportService: ReportService,
    private userService: AuthService,
    private strategySvc: SettingsService,
    private router: Router,
    private planLimitationsGuard: PlanLimitationsGuard,
    private appContext: AppContextService,
    private pluginHistoryService: PluginHistoryService,
    private timezoneService: TimezoneService
  ) {}

  ngAfterViewInit() {
    if (isPlatformBrowser(this.platformId)) {
      const container = document.querySelector('.stats-card-container');
      if (container) {
        container.addEventListener('wheel', function (e) {
          if ((e as any).deltaY !== 0) {
            e.preventDefault();
            container.scrollLeft += (e as any).deltaY;
          }
        });
      }
    }
  }

  ngOnInit() {
    // SIEMPRE iniciar loading al entrar a la ventana
    this.startLoading();

    // Cargar datos básicos
    this.loadSavedData();

    // Suscribirse a los datos del contexto
    this.subscribeToContextData();

    // Obtener datos frescos
    this.getUserData();
    this.initializeStrategies();
    this.listenGroupedTrades();
    this.fetchUserRules();
    this.checkUserAccess();

    // Cargar datos de todas las cuentas
    this.loadAllAccountsData();
  }

  private subscribeToContextData() {
    // Suscribirse a los datos del usuario
    this.appContext.currentUser$.subscribe(user => {
      this.user = user;
    });

    // Suscribirse a las cuentas del usuario - SIEMPRE cargar la primera
    this.appContext.userAccounts$.subscribe(accounts => {
      // PRIMERO: Verificar si faltan datos antes de salir
      if (accounts.length > 0) {
        const currentAccountInList = accounts[0];

        // Si tenemos la misma cuenta pero nos faltan datos, cargarlos
        if (this.currentAccount &&
            this.currentAccount.accountID === currentAccountInList.accountID &&
            (!this.balanceData || !this.stats || this.accountHistory.length === 0)) {
          this.startInternalLoading();
          this.loadSavedReportData(this.currentAccount.accountID);
          return; // Salir después de cargar
        }
      }

      // Evitar bucles infinitos - solo procesar si hay cambios reales
      if (JSON.stringify(this.accountsData) === JSON.stringify(accounts)) {
        return;
```

```
      }

      this.accountsData = accounts;
      if (accounts.length > 0) {
        // Solo procesar si la cuenta actual cambió o si no hay cuenta actual
        const newAccount = accounts[0];
        const accountChanged = !this.currentAccount ||
                               this.currentAccount.accountID !== newAccount.accountID;

        if (accountChanged) {
          this.currentAccount = newAccount;

          // Verificar si es una cuenta nueva (recién registrada)
          const isNew = this.isNewAccount(this.currentAccount);

          if (isNew) {
            // Cuenta nueva - hacer peticiones a la API
            this.startInternalLoading();
            this.fetchUserKey(this.currentAccount);
          } else {
            // Cuenta existente - SIEMPRE mostrar loading
            this.startInternalLoading();
            this.loadSavedReportData(this.currentAccount.accountID);
          }
        }
      } else {
        // Si no hay cuentas, limpiar datos y parar loading
        this.currentAccount = null;
        this.accountHistory = [];
        this.stats = undefined;
        this.balanceData = null;
        this.stopInternalLoading();
      }
    });

    // Suscribirse a las estrategias del usuario
    this.appContext.userStrategies$.subscribe(strategies => {
      this.strategies = strategies;
    });
  }

  private startLoading() {
    // Loading general solo para cuentas
    this.loading = true;

    // Timeout de seguridad para cuentas
    if (this.loadingTimeout) {
      clearTimeout(this.loadingTimeout);
    }

    this.loadingTimeout = setTimeout(() => {
      this.loading = false;
    }, 2000); // 10 segundos máximo para cuentas
  }

  private startInternalLoading() {
    // Loading interno para datos de reporte
    this.hasPendingRequests = true;
    this.statsProcessed = false;
    this.chartsRendered = false;

    // Reset account-related loading states
    this.loadingStates = {
      userData: this.loadingStates.userData,
      accounts: this.loadingStates.accounts,
      strategies: this.loadingStates.strategies,
      userKey: false,
      historyData: false,
      balanceData: false,
      config: this.loadingStates.config
```

```
    };

    // Limpiar datos temporales para evitar mostrar valores 0
    this.accountHistory = [];
    this.stats = undefined;
    // NO limpiar balanceData aquí - se mantendrá del localStorage

    // Limpiar el store para evitar mostrar datos anteriores
    this.store.dispatch(setGroupedTrades({ groupedTrades: [] }));
    this.store.dispatch(setNetPnL({ netPnL: 0 }));
    this.store.dispatch(setTradeWin({ tradeWin: 0 }));
    this.store.dispatch(setProfitFactor({ profitFactor: 0 }));
    this.store.dispatch(setAvgWnL({ avgWnL: 0 }));
    this.store.dispatch(setTotalTrades({ totalTrades: 0 }));

    // Aumentar el tiempo de loading para evitar parpadeos
    setTimeout(() => {
      this.checkIfAllDataLoaded();
    }, 800); // Esperar 1 segundo antes de verificar
  }

  private stopLoading() {
    // Solo para loading interno
    this.hasPendingRequests = false;
    this.statsProcessed = true; // Marcar que las estadísticas están procesadas
    this.chartsRendered = true; // Marcar que las gráficas están renderizadas
  }

  private stopInternalLoading() {
    // Parar loading interno
    this.hasPendingRequests = false;
    this.statsProcessed = true;
    this.chartsRendered = true;
  }

  /**
   * Verificar si una cuenta es nueva (recién registrada)
   * Una cuenta es nueva si no tiene datos guardados en localStorage
   */
  private isNewAccount(account: AccountData): boolean {
    if (!account || !isPlatformBrowser(this.platformId)) {
      return false;
    }

    try {
      // Verificar si existe datos guardados para esta cuenta
      const savedData = this.appContext.loadReportDataFromLocalStorage(account.accountID);

      // Si no hay datos guardados, es una cuenta nueva
      return !savedData || !savedData.accountHistory || !savedData.stats;
    } catch (error) {
      console.error('Error verificando si la cuenta es nueva:', error);
      // En caso de error, asumir que es nueva
      return true;
    }
  }

  private setLoadingState(key: keyof typeof this.loadingStates, value: boolean) {
    this.loadingStates[key] = value;
    this.checkIfAllDataLoaded();
  }

  private checkIfAllDataLoaded() {
    // Verificar si todos los datos críticos están cargados
    const criticalDataLoaded =
      this.loadingStates.userData &&
      this.loadingStates.accounts &&
      this.loadingStates.strategies &&
      this.loadingStates.config;

```

```
    // Si hay cuenta actual, verificar que los datos necesarios estén cargados
    // Balance e historial son independientes - no requieren ambos
    const accountDataLoaded = !this.currentAccount ||
      (this.loadingStates.userKey &&
       (this.loadingStates.historyData || this.loadingStates.balanceData));

    // Verificar que los datos estén realmente disponibles para mostrar en la UI
    const uiDataReady = this.isUIDataReady();

    // Verificar que no haya peticiones pendientes
    const noPendingRequests = !this.hasPendingRequests;

    // Verificar que las estadísticas estén completamente procesadas
    const statsReady = this.statsProcessed || !this.currentAccount;

    // Verificar que las gráficas estén completamente renderizadas
    const chartsReady = this.chartsRendered || !this.currentAccount;

    // Verificación adicional: asegurar que los datos estén realmente en localStorage
    const dataInLocalStorage = !this.currentAccount || this.isDataInLocalStorage();

    // Evitar bucles infinitos - solo procesar si no está ya procesado
    const shouldProcess = !this.statsProcessed || !this.chartsRendered;

    if (criticalDataLoaded && accountDataLoaded && uiDataReady && noPendingRequests &&
statsReady && chartsReady && dataInLocalStorage && shouldProcess) {
      // Todos los datos están cargados y listos para mostrar
      this.loadSavedReportData(this.currentAccount?.accountID || '');
    }
  }

  private isDataInLocalStorage(): boolean {
    if (!this.currentAccount || !isPlatformBrowser(this.platformId)) {
      return true;
    }

    try {
      const reportDataKey = `${this.STORAGE_KEYS.REPORT_DATA}
_${this.currentAccount.accountID}`;
      const savedReportData = localStorage.getItem(reportDataKey);
      if (savedReportData) {
        const data = JSON.parse(savedReportData);
        return data.accountHistory && data.stats && data.balanceData !== null;
      }
    } catch (error) {
      console.error('Error checking localStorage data:', error);
    }

    return false;
  }

  private isUIDataReady(): boolean {
    // Verificar que todos los datos necesarios para la UI estén disponibles

    // Si no hay cuenta, no necesitamos datos de trading
    if (!this.currentAccount) {
      return true;
    }

    // Verificar que tenemos datos de balance (puede ser 0, pero debe estar cargado)
    const hasBalanceData = this.balanceData !== null && this.balanceData !== undefined;

    // Verificar que tenemos datos de trading history (puede ser array vacío, pero debe
estar cargado)
    const hasHistoryData = Array.isArray(this.accountHistory);

    // Verificar que tenemos estadísticas completamente calculadas
    const hasStats = this.stats !== null &&
                     this.stats !== undefined &&
                     typeof this.stats.netPnl === 'number' &&
                     typeof this.stats.tradeWinPercent === 'number' &&
                     typeof this.stats.profitFactor === 'number' &&
                     typeof this.stats.totalTrades === 'number' &&
```

```
                    typeof this.stats.avgWinLossTrades === 'number';

    // Verificar que tenemos estrategias (puede ser array vacío, pero debe estar cargado)
    const hasStrategies = Array.isArray(this.strategies);

    // Para cuentas nuevas, no verificar datos reales - solo que estén cargados
    const hasProcessedData = this.statsProcessed && this.chartsRendered;

    // Balance e historial son independientes - al menos uno debe estar disponible
    const hasAccountData = hasBalanceData || hasHistoryData;

    return hasAccountData && hasStats && hasStrategies && hasProcessedData;
  }

  private hasRealChartData(): boolean {
    // Si no hay datos de trading history, las gráficas mostrarán valores por defecto
    if (!this.accountHistory || this.accountHistory.length === 0) {
      return true; // Es válido mostrar gráficas vacías si no hay datos
    }

    // Verificar que los datos de trading history tengan valores reales
    const hasRealTradingData = this.accountHistory.some(trade =>
      trade.pnl !== undefined &&
      trade.pnl !== null &&
      trade.lastModified !== undefined &&
      trade.lastModified !== null
    );

    // Verificar que las estadísticas no sean solo valores por defecto
    const hasRealStats = this.stats ? (
      this.stats.netPnl !== 0 ||
      this.stats.totalTrades > 0 ||
      this.stats.tradeWinPercent !== 0 ||
      this.stats.profitFactor !== 0
    ) : false;

    return hasRealTradingData && hasRealStats;
  }

  private loadSavedReportData(accountID: string) {
    if (!isPlatformBrowser(this.platformId) || !accountID) {
      // Si no hay accountID, parar loading interno
      this.stopInternalLoading();
      return;
    }

    try {
      const savedData = this.appContext.loadReportDataFromLocalStorage(accountID);
      if (savedData && savedData.accountHistory && savedData.stats) {
        // Simular tiempo de loading para mostrar el spinner
        setTimeout(() => {
          this.accountHistory = savedData.accountHistory;
          this.stats = savedData.stats;
          this.balanceData = savedData.balanceData;

          // Actualizar el store
          const groupedTrades = Array.isArray(savedData.accountHistory) ?
            savedData.accountHistory.map((trade: any) => ({
              ...trade,
              pnl: trade.pnl ?? 0,
              isWon: trade.isWon ?? false,
              isOpen: trade.isOpen ?? false
            })) : [];
          this.store.dispatch(setGroupedTrades({ groupedTrades }));

          // Marcar como cargados
          this.setLoadingState('userKey', true);
          this.setLoadingState('historyData', true);
          // Marcar balance como cargado si existe (incluso si es null, pero está cargado)
          this.setLoadingState('balanceData', true);
```

```
          this.hasPendingRequests = false;
        }, 800); // Esperar 2 segundos para mostrar loading
      } else {
        // No hay datos guardados, parar loading interno
        this.stopInternalLoading();
      }
    } catch (error) {
      console.error('Error cargando datos de reporte guardados:', error);
      // En caso de error, parar loading interno
      this.stopInternalLoading();
    }
  }

  // Métodos para persistencia local
  private loadSavedData() {
    if (!isPlatformBrowser(this.platformId)) return;

    try {
      // Solo cargar datos básicos para inicialización
      // Los datos de reporte se cargarán desde el contexto

      // Cargar cuentas guardadas (solo para inicialización)
      const savedAccountsData = localStorage.getItem(this.STORAGE_KEYS.ACCOUNTS_DATA);
      if (savedAccountsData) {
        this.accountsData = JSON.parse(savedAccountsData);
      }

      // Cargar cuenta actual (solo para inicialización)
      const savedCurrentAccount = localStorage.getItem(this.STORAGE_KEYS.CURRENT_ACCOUNT);
      if (savedCurrentAccount) {
        this.currentAccount = JSON.parse(savedCurrentAccount);
      }

      // Cargar datos de usuario (solo para inicialización)
      const savedUserData = localStorage.getItem(this.STORAGE_KEYS.USER_DATA);
      if (savedUserData) {
        this.user = JSON.parse(savedUserData);
      }

      if (this.currentAccount) {
        const savedAccountHistory =
localStorage.getItem(`Balance_${this.currentAccount.accountID}`);
        if (savedAccountHistory) {
          this.balanceData = JSON.parse(savedAccountHistory);
        }
      }

    } catch (error) {
      console.error('Error cargando datos guardados:', error);
      this.clearSavedData();
    }
  }

  private saveDataToStorage() {
    if (!isPlatformBrowser(this.platformId)) return;

    try {
      // Guardar datos de reporte usando el contexto
      if (this.accountHistory.length > 0 && this.stats && this.currentAccount) {
        const reportData = {
          accountHistory: this.accountHistory,
          stats: this.stats,
          balanceData: this.balanceData,
          lastUpdated: Date.now()
        };
        this.appContext.saveReportDataToLocalStorage(
          this.currentAccount.accountID,
          this.currentAccount,
          reportData
        );
      }
```

```
607
608       // Guardar cuentas
609       if (this.accountsData.length > 0) {
610         localStorage.setItem(this.STORAGE_KEYS.ACCOUNTS_DATA,
611 JSON.stringify(this.accountsData));
612       }
613
614       // Guardar cuenta actual
615       if (this.currentAccount) {
616         localStorage.setItem(this.STORAGE_KEYS.CURRENT_ACCOUNT,
617 JSON.stringify(this.currentAccount));
618       }
619
620       // Guardar datos de usuario
621       if (this.user) {
622         localStorage.setItem(this.STORAGE_KEYS.USER_DATA, JSON.stringify(this.user));
623       }
624     } catch (error) {
625       console.error('Error guardando datos:', error);
626     }
627   }
628
629
630   private clearSavedData() {
631     if (!isPlatformBrowser(this.platformId)) return;
632
633     try {
634       // Limpiar datos de reporte para la cuenta actual usando el contexto
635       if (this.currentAccount) {
636         this.appContext.clearReportDataFromLocalStorage(this.currentAccount.accountID);
637       }
638       localStorage.removeItem(this.STORAGE_KEYS.ACCOUNTS_DATA);
639       localStorage.removeItem(this.STORAGE_KEYS.CURRENT_ACCOUNT);
640       localStorage.removeItem(this.STORAGE_KEYS.USER_DATA);
641     } catch (error) {
642       console.error('Error limpiando datos guardados:', error);
643     }
644   }
645
646   private async initializeStrategies(): Promise<void> {
647     if (this.user?.id) {
648       try {
649         this.strategies = await this.strategySvc.getUserStrategyViews(this.user.id);
650         // Marcar estrategias como cargadas
651         this.setLoadingState('strategies', true);
652         // Verificar si todos los datos están listos después de cargar las estrategias
653         this.checkIfAllDataLoaded();
654       } catch (error) {
655         console.error('Error loading strategies:', error);
656         // Marcar estrategias como cargadas incluso en error
657         this.setLoadingState('strategies', true);
658         // Verificar si todos los datos están listos incluso en caso de error
659         this.checkIfAllDataLoaded();
660       }
661     } else {
662       // Si no hay usuario, marcar como cargado
663       this.setLoadingState('strategies', true);
664       // Verificar si todos los datos están listos
665       this.checkIfAllDataLoaded();
666     }
667   }
668
669   ngOnDestroy() {
670     this.updateSubscription?.unsubscribe();
671     if (this.loadingTimeout) {
672       clearTimeout(this.loadingTimeout);
673     }
674   }
675
676   getUserData() {
677     this.store.select(selectUser).subscribe({
678       next: (user) => {
```

```
        this.user = user.user;
        if (this.user) {
          // Marcar datos de usuario como cargados
          this.setLoadingState('userData', true);

          // Guardar datos de usuario en localStorage
          this.saveDataToStorage();

          // Reutilizar cuentas del contexto (ya cargadas en login)
          this.useAccountsFromContext();
        }
      },
      error: (err) => {
        console.error('Error fetching user data', err);
        this.setLoadingState('userData', true); // Marcar como cargado incluso en error
      },
    });
  }

  useAccountsFromContext() {
    // Reutilizar cuentas del contexto (ya cargadas en login)
    const contextAccounts = this.appContext.userAccounts();
    if (contextAccounts && contextAccounts.length > 0) {
      // Evitar bucles infinitos - solo procesar si hay cambios reales
      if (JSON.stringify(this.accountsData) === JSON.stringify(contextAccounts)) {
        return;
      }

      this.accountsData = contextAccounts;
      this.currentAccount = this.accountsData[0]; // Siempre la primera

      // Guardar cuentas en localStorage
      this.saveDataToStorage();

      // Marcar cuentas como cargadas
      this.setLoadingState('accounts', true);

      // SIEMPRE iniciar loading interno para mostrar loading
      this.startInternalLoading();

      // Cargar datos de la primera cuenta
      this.loadSavedReportData(this.currentAccount.accountID);
      // Lanzar carga completa de todas las cuentas y actualizar métricas por cuenta
      this.loadAllAccountsData();
    } else {
      // Si no hay cuentas en el contexto, iniciar loading y cargarlas
      this.startLoading();
      this.fetchUserAccounts();
    }
  }


  fetchUserAccounts() {
    this.userService.getUserAccounts(this.user?.id).then((accounts) => {
      if (!accounts || accounts.length === 0) {
        // No hay cuentas - usuario nuevo
        this.accountsData = [];
        this.currentAccount = null;
        this.accountHistory = [];
        this.stats = undefined;
        this.balanceData = null;

        // Marcar cuentas como cargadas
        this.setLoadingState('accounts', true);

        // Parar loading general
        this.loading = false;

        // Parar loading interno
        this.stopInternalLoading();
```

```
      } else {
        this.accountsData = accounts;
        this.currentAccount = accounts[0]; // Siempre la primera

        // Guardar cuentas en localStorage
        this.saveDataToStorage();

        // Marcar cuentas como cargadas
        this.setLoadingState('accounts', true);

        // Parar loading general
        this.loading = false;

        // Iniciar loading interno y cargar datos de la primera cuenta
        this.startInternalLoading();
        this.loadSavedReportData(this.currentAccount.accountID);
        // Lanzar carga completa de todas las cuentas y actualizar métricas por cuenta
        this.loadAllAccountsData();
      }
    }).catch((error) => {
      console.error('Error fetching user accounts:', error);
      // En caso de error, limpiar todo
      this.accountsData = [];
      this.currentAccount = null;
      this.accountHistory = [];
      this.stats = undefined;
      this.balanceData = null;

      this.setLoadingState('accounts', true);
      this.loading = false; // Parar loading general
      this.stopInternalLoading();
    });
  }

  fetchUserRules() {
    this.strategySvc
      .getStrategyConfig(this.user?.id)
      .then((data) => {
        if (data) {
          this.store.dispatch(resetConfig({ config: data }));
          this.config = this.prepareConfigDisplayData(data);
        } else {
          this.store.dispatch(resetConfig({ config: initialStrategyState }));
          this.config = this.prepareConfigDisplayData(initialStrategyState);
        }
        // Marcar configuración como cargada
        this.setLoadingState('config', true);
        // Verificar si todos los datos están listos después de cargar la configuración
        this.checkIfAllDataLoaded();
      })
      .catch((err) => {
        console.error('Error to get the config', err);
        this.store.dispatch(resetConfig({ config: initialStrategyState }));
        this.config = this.prepareConfigDisplayData(initialStrategyState);
        // Marcar configuración como cargada incluso en error
        this.setLoadingState('config', true);
        // Verificar si todos los datos están listos incluso en caso de error
        this.checkIfAllDataLoaded();
      });
  }

  onStrategyPercentageChange(percentage: number) {
    if (this.user) {
      this.updateFirebaseUserData(percentage);
    }
  }

  updateFirebaseUserData(percentage: number) {
    if (this.user) {
      const updatedUser = {
```

```
        ...this.user,
        // TODO: revisar uso anterior de best_trade, ahora pertenece a AccountData
        // TODO: revisar uso anterior de netPnl, ahora pertenece a AccountData
        lastUpdated: new Date().getTime() as unknown as Timestamp,
        // TODO: revisar uso anterior de number_trades, ahora pertenece a AccountData
        strategy_followed: percentage,
        // TODO: revisar uso anterior de profit, ahora pertenece a AccountData
        // TODO: revisar uso anterior de total_spend, ahora pertenece a AccountData
      };

      const actualYear = new Date().getFullYear();

      const requestYear = this.requestYear;
      if (actualYear === requestYear) {
        this.userService.createUser(updatedUser as unknown as User);
      }

      const monthlyReport = {
        id: this.user?.id,
        // TODO: revisar uso anterior de best_trade, ahora pertenece a AccountData
        // TODO: revisar uso anterior de netPnl, ahora pertenece a AccountData
        // TODO: revisar uso anterior de number_trades, ahora pertenece a AccountData
        // TODO: revisar uso anterior de profit, ahora pertenece a AccountData
        strategy_followed: percentage,
        // TODO: revisar uso anterior de total_spend, ahora pertenece a AccountData
        month: new Date().getMonth() + 1,
        year: new Date().getFullYear(),
      };

      this.reportService.updateMonthlyReport(
        monthlyReport as unknown as MonthlyReport
      );
    }
  }

  listenGroupedTrades() {
    this.store.select(selectGroupedTrades).subscribe({
      next: (groupedTrades) => {
        this.accountHistory = groupedTrades;
        // Calcular estadísticas inmediatamente cuando se reciben los datos
        this.updateReportStats(this.store, groupedTrades);
        // NO verificar aquí - se verificará desde updateReportStats
      },
    });
  }


  fetchUserKey(account: AccountData) {
    // Usar el servicio que ya actualiza el contexto automáticamente
    this.reportService
      .getUserKey(
        account.emailTradingAccount,
        account.brokerPassword,
        account.server
      )
      .subscribe({
        next: (key: string) => {
          this.userKey = key;
          // Marcar userKey como cargado
          this.setLoadingState('userKey', true);

          const now = new Date();
          const currentYear = now.getUTCFullYear();
          this.fromDate = Date.UTC(currentYear, 0, 1, 0, 0, 0, 0).toString();
          this.toDate = Date.UTC(
            currentYear,
            11,
            31,
            23,
            59,
```

```
            59,
            999
          ).toString();
          this.requestYear = currentYear;

          this.fetchHistoryData(
            key,
            account.accountID,
            account.accountNumber
          );

          this.store.dispatch(setUserKey({ userKey: key }));
        },
        error: (err) => {
          console.error('Error fetching user key:', err);
          this.store.dispatch(setUserKey({ userKey: '' }));
          // Marcar userKey como cargado incluso en error
          this.setLoadingState('userKey', true);
          // Marcar que no hay peticiones pendientes
          this.hasPendingRequests = false;
        },
      });
  }

  fetchHistoryData(
    key: string,
    accountId: string,
    accNum: number
  ) {
    // Solo consultar balance si no existe ya
    if (this.balanceData === null || this.balanceData === undefined) {
      this.reportService.getBalanceData(accountId, key, accNum).subscribe({
        next: (balanceData) => {
          this.balanceData = balanceData;
          this.setLoadingState('balanceData', true);
          // Verificar si todos los datos están listos después de cargar el balance
          this.checkIfAllDataLoaded();
        },
        error: (err) => {
          console.error('Error fetching balance data:', err);
          this.setLoadingState('balanceData', true);
          // Verificar si todos los datos están listos incluso en caso de error
          this.checkIfAllDataLoaded();
        },
      });
    } else {
      // Si ya existe balanceData, marcar como cargado
      this.setLoadingState('balanceData', true);
      this.checkIfAllDataLoaded();
    }

    // Solo hacer petición al trading history (la principal)
    this.reportService
      .getHistoryData(accountId, key, accNum)
      .subscribe({
        next: (groupedTrades: GroupedTradeFinal[]) => {
          // Reemplazar en lugar de acumular para evitar duplicados
          this.store.dispatch(
            setGroupedTrades({
              groupedTrades: groupedTrades,
            })
          );

          // Calcular estadísticas inmediatamente después de recibir los datos
          this.updateReportStats(this.store, groupedTrades);

          // Guardar datos en el contexto y localStorage por cuenta DESPUÉS de calcular stats
          if (this.currentAccount) {
            // Esperar a que las estadísticas estén calculadas
            // Guardar en el contexto
```

```
            const contextData = {
              accountHistory: groupedTrades,
              stats: this.stats,
              balanceData: this.balanceData,
              lastUpdated: Date.now()
            };
            this.appContext.setTradingHistoryForAccount(this.currentAccount!.id,
contextData);
            // Guardar datos en localStorage después de recibir respuesta exitosa
            this.saveDataToStorage();

            // Marcar history data como cargado DESPUÉS de guardar todo
            this.setLoadingState('historyData', true);
            // Marcar que no hay peticiones pendientes
            this.hasPendingRequests = false;
          } else {
            // Si no hay cuenta actual, marcar como cargado inmediatamente
            this.setLoadingState('historyData', true);
            this.hasPendingRequests = false;
          }
        },
        error: (err) => {
          console.error('Error fetching history data:', err);
          this.store.dispatch(setGroupedTrades({ groupedTrades: [] }));
          // Marcar history data como cargado incluso en error
          this.setLoadingState('historyData', true);
          // Marcar que no hay peticiones pendientes
          this.hasPendingRequests = false;
        },
      });
  }

  updateReportStats(store: Store, groupedTrades: GroupedTradeFinal[]) {
    // Normalizar todos los trades - asignar valores por defecto cuando falten
    const normalizedTrades = groupedTrades.map(trade => ({
      ...trade,
      pnl: trade.pnl ?? 0, // Si no hay PnL, usar 0
      entryPrice: trade.avgPrice ?? 0, // Usar avgPrice como entryPrice
      exitPrice: trade.avgPrice ?? 0, // Usar avgPrice como exitPrice
      buy_price: trade.side === 'buy' ? trade.price : '0', // Precio de compra si es buy
      sell_price: trade.side === 'sell' ? trade.price : '0', // Precio de venta si es sell
      quantity: Number(trade.qty) ?? 0 // Usar qty como quantity
    }));

    // Usar las funciones de utilidad existentes
    this.stats = {
      netPnl: calculateNetPnl(normalizedTrades),
      tradeWinPercent: calculateTradeWinPercent(normalizedTrades),
      profitFactor: calculateProfitFactor(normalizedTrades),
      avgWinLossTrades: calculateAvgWinLossTrades(normalizedTrades),
      totalTrades: calculateTotalTrades(normalizedTrades),
      activePositions: groupedTrades.filter(trade => trade.isOpen === true).length
    };

    // Actualizar el store con las estadísticas calculadas
    store.dispatch(setNetPnL({ netPnL: this.stats?.netPnl || 0 }));
    store.dispatch(setTradeWin({ tradeWin: this.stats?.tradeWinPercent || 0 }));
    store.dispatch(setProfitFactor({ profitFactor: this.stats?.profitFactor || 0 }));
    store.dispatch(setAvgWnL({ avgWnL: this.stats?.avgWinLossTrades || 0 }));
    store.dispatch(setTotalTrades({ totalTrades: this.stats?.totalTrades || 0 }));

    // Guardar stats actualizados en localStorage
    this.saveDataToStorage();

    // Marcar que las estadísticas están procesadas
    this.statsProcessed = true;
    this.chartsRendered = true;

    // Esperar un poco antes de verificar para evitar recargas múltiples
    setTimeout(() => {
```

```
        this.checkIfAllDataLoaded();
      }, 800);
    }

    prepareConfigDisplayData(strategyState: StrategyState) {
      return this.transformStrategyStateToDisplayData(strategyState);
    }

    transformStrategyStateToDisplayData(
      strategyState: StrategyState
    ): displayConfigData[] {
      const newConfig: displayConfigData[] = [];

      Object.entries(strategyState).forEach(([key, value]) => {
        if (value.type) {
          let title = '';

          if (value.type === RuleType.MAX_DAILY_TRADES) {
            title = 'Max Daily Trades';
          } else if (value.type === RuleType.RISK_REWARD_RATIO) {
            title = `${value.riskRewardRatio} Risk Reward`;
          } else if (value.type === RuleType.MAX_RISK_PER_TRADE) {
            title = `Limit my Risk per Trade: $${value.maxRiskPerTrade}`;
          } else if (value.type === RuleType.DAYS_ALLOWED) {
            const abbreviationsMap: { [key: string]: string } = {
              Monday: 'Mon',
              Tuesday: 'Tues',
              Wednesday: 'Wed',
              Thursday: 'Thurs',
              Friday: 'Fri',
              Saturday: 'Sat',
              Sunday: 'Sun',
            };

            const formattedText = value.tradingDays
              .map((day: string) => abbreviationsMap[day] ?? day)
              .join(', ');

            title = `Only allow trades on: ${formattedText}.`;
          } else if (value.type === RuleType.ASSETS_ALLOWED) {
            const formattedText = value.assetsAllowed.join(', ');
            title = `Only allow trades on the following assets: ${formattedText}`;
          } else if (value.type === RuleType.TRADING_HOURS) {
            const abbreviation = moment.tz(value.timezone).zoneAbbr();
            title = `Only allow trades between: ${value.tradingOpenTime} -
${value.tradingCloseTime} ${abbreviation}`;
          }

          newConfig.push({
            title,
            type: value.type,
            isActive: value.isActive,
          });
        }
      });

      return newConfig.sort((a, b) => {
        if (a.isActive && !b.isActive) return -1;
        if (!a.isActive && b.isActive) return 1;
        return 0;
      });
    }

    onYearChange($event: string) {
      this.startLoading();
      this.fromDate = Date.UTC(Number($event), 0, 1, 0, 0, 0, 0).toString();
      this.toDate = Date.UTC(Number($event), 11, 31, 23, 59, 59, 999).toString();
      this.requestYear = Number($event);

      // Reset account-related loading states
      this.setLoadingState('userKey', false);
```

```
    this.setLoadingState('historyData', false);
    this.setLoadingState('balanceData', false);

    if (this.user) {
      this.fetchUserAccounts();
    }
  }

  // Account management methods
  getCurrentAccountName(): string {
    return this.currentAccount?.accountName || 'No Account Selected';
  }

  getCurrentAccountServer(): string {
    return this.currentAccount?.server || 'No Server';
  }

  async getCurrentAccountPlan(): Promise<string> {
    return await this.getAccountPlan(this.currentAccount);
  }

  async getAccountPlan(account: AccountData | null): Promise<string> {
    if (!account || !this.user?.id) return 'Free';

    // Use the guard to get the real plan information
    return await this.getUserPlanName();
  }

  private async getUserPlanName(): Promise<string> {
    if (!this.user?.id) return 'Free';

    try {
      const limitations = await this.planLimitationsGuard.checkUserLimitations(this.user.id);
      return limitations.planName;
    } catch (error) {
      console.error('Error getting user plan name:', error);
      return 'Free';
    }
  }

  toggleAccountDropdown() {
    this.showAccountDropdown = !this.showAccountDropdown;
  }

  selectAccount(account: AccountData) {
    this.currentAccount = account;
    this.showAccountDropdown = false;

    // Guardar cuenta seleccionada en localStorage
    this.saveDataToStorage();

    // SIEMPRE iniciar loading interno para mostrar loading
    this.startInternalLoading();

    // Reset account-related loading states
    this.setLoadingState('userKey', false);
    this.setLoadingState('historyData', false);
    this.setLoadingState('balanceData', false);

    // Limpiar datos anteriores
    this.store.dispatch(setGroupedTrades({ groupedTrades: [] }));

    // Verificar si existe localStorage para esta cuenta
    const savedData = this.appContext.loadReportDataFromLocalStorage(account.accountID);

    if (savedData && savedData.accountHistory && savedData.stats) {
      // Si existe en localStorage, cargar directamente con loading
      this.loadSavedReportData(account.accountID);
    } else {
      // Si no existe, hacer peticiones a la API
```

```
      this.fetchUserKey(account);
    }
  }

  goToEditStrategy() {
    this.router.navigate(['/edit-strategy']);
  }

  async exportAllData() {
    const csvData = await this.generateAllReportsCSV();
    this.downloadCSV(csvData, `my-reports-${new Date().toISOString().split('T')[0]}.csv`);
  }

  async generateAllReportsCSV(): Promise<string> {
    const headers = [
      'Date',
      'Account Name',
      'Plan',
      'Net P&L',
      'Trades Count',
      'Win Percentage',
      'Strategy Followed',
      'Profit Factor',
      'Avg Win/Loss Trades'
    ];
    const rows = [headers.join(',')];

    // Add summary data
    const currentPlan = await this.getCurrentAccountPlan();
    const summaryRow = [
      new Date().toISOString().split('T')[0],
      this.getCurrentAccountName(),
      currentPlan,
      this.stats?.netPnl?.toFixed(2) || '0',
      this.stats?.totalTrades?.toString() || '0',
      `${this.stats?.tradeWinPercent?.toFixed(1) || '0'}%`,
      'Yes',
      this.stats?.profitFactor?.toFixed(2) || '0',
      this.stats?.avgWinLossTrades?.toFixed(2) || '0'
    ];
    rows.push(summaryRow.join(','));

    // Add detailed trade data
    this.accountHistory.forEach(trade => {
      const tradeDate = new Date(Number(trade.lastModified)).toISOString().split('T')[0];
      const tradeRow = [
        tradeDate,
        this.getCurrentAccountName(),
        currentPlan, // Use the same plan for all trades
        (trade.pnl || 0).toFixed(2), // P&L calculado correctamente
        '1',
        trade.pnl && trade.pnl > 0 ? '100' : '0', // Win percentage basado en P&L real
        'Yes',
        '1.00',
        (trade.pnl || 0).toFixed(2) // P&L neto (mismo que P&L ya que no hay fees en el cálculo)
      ];
      rows.push(tradeRow.join(','));
    });

    return rows.join('\n');
  }

  downloadCSV(csvData: string, filename: string) {
    const blob = new Blob([csvData], { type: 'text/csv;charset=utf-8;' });
    const link = document.createElement('a');

    if (link.download !== undefined) {
      const url = URL.createObjectURL(blob);
      link.setAttribute('href', url);
      link.setAttribute('download', filename);
```

```
      link.style.visibility = 'hidden';
      document.body.appendChild(link);
      link.click();
      document.body.removeChild(link);
    }
  }

  // Navegar a trading accounts
  navigateToTradingAccounts() {
    this.router.navigate(['/trading-accounts']);
  }

  // Método para recargar cuentas
  refreshAccounts() {
    if (this.user) {
      this.fetchUserAccounts();
    }
  }

  // Método para recargar datos manualmente
  reloadData() {
    this.showReloadButton = false;
    this.startLoading();

    // Limpiar datos anteriores y localStorage
    this.store.dispatch(setGroupedTrades({ groupedTrades: [] }));
    this.accountHistory = [];
    this.stats = undefined;
    this.balanceData = null;
    this.statsProcessed = false;
    this.chartsRendered = false;
    this.clearSavedData();

    // Reset all loading states
    this.setLoadingState('userData', false);
    this.setLoadingState('accounts', false);
    this.setLoadingState('strategies', false);
    this.setLoadingState('userKey', false);
    this.setLoadingState('historyData', false);
    this.setLoadingState('balanceData', false);
    this.setLoadingState('config', false);

    // Reiniciar el proceso de carga
    if (this.user) {
      this.useAccountsFromContext();
    } else {
      this.getUserData();
    }
  }

  // Check user access and show blocking modal if needed
  async checkUserAccess() {
    if (!this.user?.id) return;

    try {
      const accessCheck = await
this.planLimitationsGuard.checkReportAccessWithModal(this.user.id);
      // Only show blocking modal if user has trading accounts (not first-time user with
plan) if (!accessCheck.canAccess && accessCheck.modalData && this.accountsData.length > 0) {
        this.planLimitationModal = accessCheck.modalData;
      }
    } catch (error) {
      console.error('Error checking user access:', error);
    }
  }

  // Plan limitation modal methods
  onClosePlanLimitationModal() {
    this.planLimitationModal.showModal = false;
  }
```

```
1307
1308    // ===== MÉTODOS DE CARGA DE TODAS LAS CUENTAS =====
1309
1310    /**
1311     * Cargar datos de todas las cuentas del usuario
1312     */
1313    async loadAllAccountsData() {
1314      if (!this.user?.id || this.accountsData.length === 0) {
1315        return;
1316      }
1317
1318      try {
1319        // Cargar datos de todas las cuentas en paralelo
1320        const loadPromises = this.accountsData.map(account =>
1321          this.loadAccountData(account)
1322        );
1323
1324        await Promise.all(loadPromises);
1325
1326        // Calcular métricas globales del usuario
1327        await this.calculateAndUpdateUserMetrics();
1328
1329        // Calcular strategy_followed basado en todas las cuentas
1330        await this.calculateAndUpdateStrategyFollowed();
1331
1332        // NUEVO: Calcular y actualizar métricas por cuenta (netPnl, profit factor y bestTrade)
1333        await this.calculateAndUpdateAccountsMetrics();
1334
1335      } catch (error) {
1336        console.error('Error loading all accounts data:', error);
1337      }
1338    }
1339
1340    /**
1341     * Cargar datos de una cuenta específica
1342     */
1343    private async loadAccountData(account: AccountData): Promise<void> {
1344      try {
1345        // Obtener userKey
1346        const userKey = await this.reportService.getUserKey(
1347          account.emailTradingAccount,
1348          account.brokerPassword,
1349          account.server
1350        ).toPromise();
1351
1352        if (!userKey) {
1353          console.warn(`No se pudo obtener userKey para cuenta ${account.accountID}`);
1354          return;
1355        }
1356
1357        // Obtener trading history
1358        const tradingHistory = await this.reportService.getHistoryData(
1359          account.accountID,
1360          userKey,
1361          account.accountNumber
1362        ).toPromise();
1363
1364        // Obtener balance data
1365        const balanceData = await this.reportService.getBalanceData(
1366          account.accountID,
1367          userKey,
1368          account.accountNumber
1369        ).toPromise();
1370
1371        // Guardar en localStorage
1372        this.saveAccountDataToLocalStorage(account.accountID, {
1373          accountHistory: tradingHistory || [],
1374          balanceData: balanceData,
1375          lastUpdated: Date.now()
1376        });
```

```
    } catch (error) {
      console.error(`Error loading data for account ${account.accountID}:`, error);
    }
  }

  /**
   * Calcular y actualizar métricas globales del usuario
   */
  private async calculateAndUpdateUserMetrics(): Promise<void> {
    if (!this.user?.id) return;

    try {
      // Recopilar todos los trades de todas las cuentas
      const allTrades: any[] = [];
      let globalProfitFactor = 0;

      for (const account of this.accountsData) {
        const accountData = this.loadAccountDataFromLocalStorage(account.accountID);
        if (accountData && accountData.accountHistory) {
          allTrades.push(...accountData.accountHistory);
        }
      }

      if (allTrades.length > 0) {
        // Calcular profit_factor global (usando la misma lógica que en la ventana)
        globalProfitFactor = this.calculateProfitFactor(allTrades);

        // Calcular best_trade global (mejor trade de todas las cuentas)
        const bestTrade = this.calculateGlobalBestTrade(allTrades);

        // Actualizar usuario en Firebase con las métricas globales
        const updatedUser = {
          ...this.user,
          profit: globalProfitFactor, // profit = profit_factor
          best_trade: bestTrade, // best_trade = mejor trade de todas las cuentas
          lastUpdated: new Date().getTime() as unknown as Timestamp
        };

        await this.userService.createUser(updatedUser as unknown as User);
      }

    } catch (error) {
      console.error('Error calculating user metrics:', error);
    }
  }

  /**
   * Calcular el mejor trade global de todas las cuentas
   */
  private calculateGlobalBestTrade(allTrades: any[]): number {
    if (!allTrades || allTrades.length === 0) return 0;

    // Normalizar trades
    const normalizedTrades = allTrades.map(trade => ({
      ...trade,
      pnl: trade.pnl ?? 0
    }));

    // Encontrar el trade con mayor ganancia o pérdida absoluta
    const bestTrade = normalizedTrades.reduce((best, trade) => {
      const currentAbs = Math.abs(trade.pnl);
      const bestAbs = Math.abs(best);
      return currentAbs > bestAbs ? trade.pnl : best;
    }, 0);

    return Math.round(bestTrade * 100) / 100;
  }

  /**
```

```
    * Calcular porcentaje de ganancia
    */
  private calculateWinPercent(trades: any[]): number {
    if (trades.length === 0) return 0;
    const winningTrades = trades.filter(trade => trade.pnl > 0).length;
    return Math.round((winningTrades / trades.length) * 100 * 100) / 100;
  }

  /**
   * Calcular profit factor
   */
  private calculateProfitFactor(trades: any[]): number {
    const totalGains = trades
      .filter(t => t.pnl > 0)
      .reduce((sum, t) => sum + t.pnl, 0);

    const totalLosses = Math.abs(trades
      .filter(t => t.pnl < 0)
      .reduce((sum, t) => sum + t.pnl, 0));

    if (totalLosses === 0) {
      return totalGains > 0 ? 999.99 : 0;
    }

    return Math.round((totalGains / totalLosses) * 100) / 100;
  }

  /**
   * Guardar datos de cuenta en localStorage
   */
  private saveAccountDataToLocalStorage(accountID: string, data: any): void {
    try {
      const key = `tradeSwitch_reportData_${accountID}`;
      localStorage.setItem(key, JSON.stringify(data));
    } catch (error) {
      console.error('Error saving account data to localStorage:', error);
    }
  }


  /**
   * Calcular y actualizar strategy_followed en el usuario
   * NUEVA LÓGICA: Verifica cada trade individual y si el plugin estuvo activo en el momento
exacto del trade (con hora)
   */
  private async calculateAndUpdateStrategyFollowed(): Promise<void> {
    if (!this.user?.id) return;

    try {
      // 1. Obtener plugin history del usuario primero
      const pluginHistoryArray = await
this.pluginHistoryService.getPluginUsageHistory(this.user.id);
      if (pluginHistoryArray.length === 0) {
        // No hay plugin history, asumir 0%
        const updatedUser = {
          ...this.user,
          strategy_followed: 0,
          lastUpdated: new Date().getTime() as unknown as Timestamp
        };
        await this.userService.createUser(updatedUser as unknown as User);
        return;
      }

      const pluginHistory = pluginHistoryArray[0];

      // 2. Contar trades con plugin activo vs total de trades (validando hora exacta)
      let totalTrades = 0;
      let tradesWithActivePlugin = 0;

      for (const account of this.accountsData) {
        const accountData = this.loadAccountDataFromLocalStorage(account.accountID);
```

```
        if (accountData && accountData.accountHistory) {
          for (const trade of accountData.accountHistory) {
            if (trade.createdDate) {
              totalTrades++;
              // Convertir fecha del trade a UTC (con hora completa)
              const tradeDate =
this.timezoneService.convertTradeDateToUTC(trade.createdDate);
              // Verificar si el plugin estaba activo en el momento exacto de este trade
              if (this.wasPluginActiveAtTime(tradeDate, pluginHistory)) {
                tradesWithActivePlugin++;
              }
            }
          }
        }
      }

      if (totalTrades === 0) {
        // No hay trades, porcentaje es 0
        const updatedUser = {
          ...this.user,
          strategy_followed: 0,
          lastUpdated: new Date().getTime() as unknown as Timestamp
        };
        await this.userService.createUser(updatedUser as unknown as User);
        return;
      }

      // 3. Calcular porcentaje: (trades con plugin activo / total trades) * 100
      const strategyFollowedPercent = totalTrades > 0
        ? Math.round((tradesWithActivePlugin / totalTrades) * 100 * 10) / 10
        : 0;

      // 4. Actualizar usuario en Firebase
      const updatedUser = {
        ...this.user,
        strategy_followed: strategyFollowedPercent,
        lastUpdated: new Date().getTime() as unknown as Timestamp
      };

      await this.userService.createUser(updatedUser as unknown as User);

    } catch (error) {
      console.error('Error calculating strategy_followed:', error);
    }
  }

  /**
   * Verificar si el plugin estaba activo en un momento específico (con hora exacta)
   * @param tradeDate Fecha y hora del trade en UTC
   * @param pluginHistory Plugin history con dateActive y dateInactive
   */
  private wasPluginActiveAtTime(tradeDate: Date, pluginHistory: PluginHistory): boolean {
    if (!pluginHistory.dateActive || !pluginHistory.dateInactive) {
      return false;
    }

    const dateActive = pluginHistory.dateActive;
    const dateInactive = pluginHistory.dateInactive;

    // Obtener timestamp del trade en UTC
    const tradeTimestamp = tradeDate.getTime();

    // Si dateActive tiene más elementos que dateInactive, está activo desde la última fecha
activa hasta ahora
    if (dateActive.length > dateInactive.length) {
      const lastActiveDate = this.timezoneService.convertToUTC(dateActive[dateActive.length
- 1]);const lastActiveTimestamp = lastActiveDate.getTime();

      // El trade debe ser >= a la última fecha/hora activa
      return tradeTimestamp >= lastActiveTimestamp;
    }
```

```
1587
1588      // Si tienen la misma cantidad, hay pares de activación/desactivación
1589      // Verificar si el timestamp del trade está dentro de algún rango activo [dateActive[i],
1590  dateInactive[i]]
1591        for (let i = 0; i < dateActive.length; i++) {
1592          const activeDate = this.timezoneService.convertToUTC(dateActive[i]);
1593          const inactiveDate = this.timezoneService.convertToUTC(dateInactive[i]);
1594
1595          const activeTimestamp = activeDate.getTime();
1596          const inactiveTimestamp = inactiveDate.getTime();
1597
1598          // El trade debe estar >= activeTimestamp y < inactiveTimestamp (rango [active,
1599  inactive))
1600          if (tradeTimestamp >= activeTimestamp && tradeTimestamp < inactiveTimestamp) {
1601            return true;
1602          }
1603        }
1604
1605        return false;
1606      }
1607
1608      /**
1609       * Cargar datos de cuenta desde localStorage
1610       */
1611      private loadAccountDataFromLocalStorage(accountID: string): any {
1612        try {
1613          const key = `tradeSwitch_reportData_${accountID}`;
1614          const data = localStorage.getItem(key);
1615          return data ? JSON.parse(data) : null;
1616        } catch (error) {
1617          console.error('Error loading account data from localStorage:', error);
1618          return null;
1619        }
1620      }
1621
1622      // ===== MÉTODOS DE REFRESH =====
1623
1624      /**
1625       * Refrescar datos de la cuenta actual
1626       */
1627      refreshCurrentAccountData() {
1628        if (!this.currentAccount) {
1629          console.warn('No hay cuenta seleccionada para refrescar');
1630          return;
1631        }
1632
1633        // Iniciar loading interno
1634        this.startInternalLoading();
1635
1636        // Reset account-related loading states
1637        this.setLoadingState('userKey', false);
1638        this.setLoadingState('historyData', false);
1639        this.setLoadingState('balanceData', false);
1640
1641        // Limpiar datos anteriores COMPLETAMENTE
1642        this.store.dispatch(setGroupedTrades({ groupedTrades: [] }));
1643        this.accountHistory = [];
1644        this.stats = undefined;
1645        this.balanceData = null;
1646        this.statsProcessed = false;
1647        this.chartsRendered = false;
1648
1649        // Limpiar datos guardados de la cuenta actual del contexto
1650        this.appContext.clearTradingHistoryForAccount(this.currentAccount.accountID);
1651
1652        // Cargar datos frescos de la cuenta actual
1653        this.loadAccountData(this.currentAccount).then(() => {
1654          // Después de cargar, recalcular métricas globales del usuario
1655          this.calculateAndUpdateUserMetrics();
1656          // Recalcular strategy_followed
1657          this.calculateAndUpdateStrategyFollowed();
1658          // Recalcular métricas de la cuenta activa
```

```
      this.calculateAndUpdateAccountsMetrics([this.currentAccount!]);
    });
  }

  /**
   * Calcular y actualizar métricas por cuenta (AccountData)
   * - netPnl: suma de pnl
   * - profit: profit factor de la cuenta
   * - bestTrade: mayor |pnl|
   * Si no hay users/cuentas cargadas, no hace nada
   */
  private async calculateAndUpdateAccountsMetrics(targetAccounts?: AccountData[]): Promise<void> {
    const accounts = targetAccounts || this.accountsData;
    if (!accounts || accounts.length === 0) return;

    try {
      for (const account of accounts) {
        const local = this.loadAccountDataFromLocalStorage(account.accountID);
        const trades = Array.isArray(local?.accountHistory) ? local.accountHistory : [];

        if (trades.length === 0) {
          await this.userService.updateAccount(account.id, {
            ...account,
            netPnl: 0,
            profit: 0,
            bestTrade: 0,
          } as AccountData);
          continue;
        }

        // Normalizar PnL
        const normalized = trades.map((t: any) => ({ ...t, pnl: t.pnl ?? 0 }));
        const netPnl = calculateNetPnl(normalized);
        const profitFactor = this.calculateProfitFactor(normalized);
        const bestTrade = this.calculateGlobalBestTrade(normalized); // reutilizamos el cálculo de mejor trade
        await this.userService.updateAccount(account.id, {
          ...account,
          netPnl: Math.round(netPnl * 100) / 100,
          profit: Math.round(profitFactor * 100) / 100,
          bestTrade: Math.round(bestTrade * 100) / 100,
        } as AccountData);
      }
    } catch (error) {
      console.error('Error updating account metrics:', error);
    }
  }
}
```

Ø=ÜÁ features\report\components\calendar

Ø=ÜÄ features\report\components\calendar\calendar.component.ts

```
import { CommonModule } from '@angular/common';
import {
  Component,
  Input,
  Output,
  EventEmitter,
  SimpleChanges,
} from '@angular/core';
import {
  CalendarDay,
```

```
  GroupedTradeFinal,
  PluginHistoryRecord,
} from '../../models/report.model';
import { ReportService } from '../../service/report.service';
import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';
import { TradesPopupComponent } from '../trades-popup/trades-popup.component';
import { ConfigurationOverview } from '../../../strategy/models/strategy.model';
import { PluginHistoryService, PluginHistory } from '../../../../shared/services/plugin-history.service';
import { AppContextService } from '../../../../shared/context';
import { TradeLockerApiService } from '../../../../shared/services/tradelocker-api.service';
import { TimezoneService } from '../../../../shared/services/timezone.service';

@Component({
  selector: 'app-calendar',
  templateUrl: './calendar.component.html',
  styleUrls: ['./calendar.component.scss'],
  standalone: true,
  imports: [CommonModule, TradesPopupComponent],
})
export class CalendarComponent {
  @Input() groupedTrades!: GroupedTradeFinal[];
  @Output() strategyFollowedPercentageChange = new EventEmitter<number>();
  @Input() strategies!: ConfigurationOverview[];
  @Input() userId!: string; // Necesario para obtener el plugin history

  calendar: CalendarDay[][] = [];
  currentDate!: Date;
  selectedMonth!: Date;

  // Popup properties
  showTradesPopup = false;
  selectedDay: CalendarDay | null = null;

  // Plugin history properties
  pluginHistory: PluginHistory | null = null;

  constructor(
    private reportSvc: ReportService,
    private pluginHistoryService: PluginHistoryService,
    private appContext: AppContextService,
    private tradeLockerApiService: TradeLockerApiService,
    private timezoneService: TimezoneService
  ) {}
  private numberFormatter = new NumberFormatterService();

  ngOnChanges(changes: SimpleChanges) {
    this.currentDate = new Date();
    this.selectedMonth = new Date(this.currentDate.getFullYear(),
this.currentDate.getMonth());
    // Obtener userId desde el contexto de autenticación
    this.loadUserIdAndInitialize();

    // Procesar trades para obtener nombres de instrumentos
    this.processTradesForCalendar();

    this.generateCalendar(this.selectedMonth);

    if (this.strategies && this.strategies.length > 0) {
      this.getPercentageStrategyFollowedLast30Days();
    }
  }

  emitStrategyFollowedPercentage(value: number): void {
    this.strategyFollowedPercentageChange.emit(value);
  }

  /**
   * Obtener userId desde el contexto de autenticación y inicializar
   */
  private async loadUserIdAndInitialize() {
```

```
    try {
      // Obtener el usuario actual desde el contexto
      const currentUser = this.appContext.currentUser();

      if (currentUser && currentUser.id) {
        this.userId = currentUser.id;

        // Cargar plugin history con el userId obtenido
        await this.loadPluginHistory();
      }
    } catch (error) {
      console.error('Error obteniendo userId desde contexto:', error);
    }
  }

  /**
   * Convertir Timestamp de Firestore o ISO 8601 string a Date
   */
  private convertFirestoreTimestamp(timestamp: any): Date {
    // Si es string ISO 8601 (formato prioritario)
    if (typeof timestamp === 'string') {
      return new Date(timestamp);
    }
    // Fallback: Firestore Timestamp (legacy)
    if (timestamp && typeof timestamp === 'object' && 'seconds' in timestamp) {
      return new Date(timestamp.seconds * 1000 + (timestamp.nanoseconds || 0) / 1000000);
    }
    // Fallback final
    return new Date(timestamp);
  }

  /**
   * MEJORA: Convertir fecha a UTC considerando zona horaria del usuario
   * MÉTODO ESPECÍFICO: Para fechas de trades del servidor
   */
  private convertToUTCWithTimezone(date: Date | string | number): Date {
    try {
      // Usar el método específico para fechas de trades
      return this.timezoneService.convertTradeDateToUTC(date);
    } catch (error) {
      console.error('Error convirtiendo fecha a UTC:', error);
      // Fallback: conversión básica
      return new Date(date);
    }
  }

  /**
   * Procesar trades para obtener nombres de instrumentos
   */
  private async processTradesForCalendar() {
    if (!this.groupedTrades || this.groupedTrades.length === 0) {
      return;
    }

    // Verificar si los trades ya tienen nombres de instrumentos correctos
    const firstTrade = this.groupedTrades[0];
    const needsProcessing = !firstTrade.instrument ||
                            firstTrade.instrument === firstTrade.tradableInstrumentId ||
                            firstTrade.instrument === '' ||
                            firstTrade.instrument === 'Cargando...';

    if (!needsProcessing) {
      return; // Ya están procesados
    }

    try {
      // Obtener instrumentos únicos (optimización: solo una petición por combinación única)
      const uniqueInstruments = new Map<string, { tradableInstrumentId: string, routeId:
string }>();
      this.groupedTrades.forEach(trade => {
```

```
        if (trade.tradableInstrumentId && trade.routeId) {
          const key = `${trade.tradableInstrumentId}-${trade.routeId}`;
          if (!uniqueInstruments.has(key)) {
            uniqueInstruments.set(key, {
              tradableInstrumentId: trade.tradableInstrumentId,
              routeId: trade.routeId
            });
          }
        }
      });

      // Establecer "Cargando..." como valor inicial para todos los trades
      this.groupedTrades.forEach(trade => {
        if (trade.tradableInstrumentId && trade.routeId) {
          trade.instrument = 'Cargando...';
        }
      });

      // Obtener detalles de instrumentos (una sola petición por combinación única)
      const instrumentDetailsMap = new Map<string, { lotSize: number, name: string }>();

      for (const [key, instrument] of uniqueInstruments) {
        try {
          // Obtener el token real del usuario desde el contexto
          const userKey = await this.getUserToken();
          if (!userKey) {
            throw new Error('Token de usuario no disponible');
          }

          const instrumentDetails = await this.reportSvc.getInstrumentDetails(
            userKey, // Token real del usuario
            instrument.tradableInstrumentId,
            instrument.routeId,
            1
          ).toPromise();

          if (instrumentDetails) {
            instrumentDetailsMap.set(key, {
              lotSize: instrumentDetails.lotSize || 1,
              name: instrumentDetails.name || instrument.tradableInstrumentId
            });
          } else {
            instrumentDetailsMap.set(key, {
              lotSize: 1,
              name: instrument.tradableInstrumentId
            });
          }

        } catch (error) {
          console.warn(`Error obteniendo detalles del instrumento ${key}:`, error);
          instrumentDetailsMap.set(key, {
            lotSize: 1,
            name: instrument.tradableInstrumentId
          });
        }
      }

      // Actualizar trades con nombres de instrumentos (aplicar a todos los trades con la misma combinación)
      this.groupedTrades.forEach(trade => {
        if (trade.tradableInstrumentId && trade.routeId) {
          const key = `${trade.tradableInstrumentId}-${trade.routeId}`;
          const instrumentDetails = instrumentDetailsMap.get(key);

          if (instrumentDetails) {
            trade.instrument = instrumentDetails.name;
          } else {
            trade.instrument = trade.tradableInstrumentId; // Fallback al ID
          }
        }
      });
```

```
    } catch (error) {
      console.error('Error procesando trades para calendario:', error);
      // En caso de error, establecer fallback
      this.groupedTrades.forEach(trade => {
        if (trade.tradableInstrumentId && trade.routeId) {
          trade.instrument = trade.tradableInstrumentId;
        }
      });
    }
  }

  /**
   * Obtener el token del usuario desde el store
   */
  private async getUserToken(): Promise<string | null> {
    try {
      // Obtener la cuenta actual del contexto
      const accounts = this.appContext.userAccounts();
      if (!accounts || accounts.length === 0) {
        return null;
      }

      const currentAccount = accounts[0]; // Tomar la primera cuenta

      // Usar el TradeLockerApiService para obtener el token
      const userKey = await this.tradeLockerApiService.getUserKey(
        currentAccount.emailTradingAccount,
        currentAccount.brokerPassword,
        currentAccount.server
      ).toPromise();

      if (userKey) {
        return userKey;
      }

      return null;
    } catch (error) {
      console.error('Error obteniendo token del usuario:', error);
      return null;
    }
  }

  /**
   * Cargar plugin history para el usuario
   */
  async loadPluginHistory() {
    try {
      const pluginHistoryArray = await
this.pluginHistoryService.getPluginUsageHistory(this.userId);
      if (pluginHistoryArray.length > 0) {
        this.pluginHistory = pluginHistoryArray[0];
      } else {
        this.pluginHistory = null;
      }
    } catch (error) {
      console.error('Error loading plugin history:', error);
      this.pluginHistory = null;
    }
  }

  /**
   * Determinar qué estrategia se siguió en una fecha específica
   * NUEVA LÓGICA: Asociar trades con estrategias basándose en fechas exactas
   * @param tradeDate - Fecha y hora exacta del trade a validar
   * @returns nombre de la estrategia seguida o null si no se siguió ninguna
   */
  getStrategyFollowedOnDate(tradeDate: Date): string | null {
    // PASO 1: Verificar si el plugin estaba activo en la fecha/hora exacta del trade
    const pluginActiveRange = this.getPluginActiveRange(tradeDate);
    if (!pluginActiveRange) {
```

```
      return null; // Plugin no estaba activo
    }

    // PASO 2: Buscar estrategias que incluyan la fecha/hora exacta del trade
    const activeStrategy = this.getActiveStrategyAtTime(tradeDate);
    if (!activeStrategy) {
      return null; // No había estrategia activa en ese momento
    }

    return activeStrategy;
  }

  /**
   * PASO 1: Determinar si el plugin estaba activo en la fecha/hora exacta del trade
   * MEJORA: Usar conversión UTC para comparaciones precisas
   * @param tradeDate - Fecha y hora exacta del trade
   * @returns rango activo del plugin o null si no estaba activo
   */
  private getPluginActiveRange(tradeDate: Date): { start: Date, end: Date } | null {
    if (!this.pluginHistory || !this.pluginHistory.dateActive || !
this.pluginHistory.dateInactive) {
      return null;
    }

    const dateActive = this.pluginHistory.dateActive;
    const dateInactive = this.pluginHistory.dateInactive;
    const now = new Date();

    // MEJORA: Convertir fecha del trade a UTC para comparación precisa
    const tradeDateUTC = this.convertToUTCWithTimezone(tradeDate);


    // Crear rangos de actividad del plugin
    const activeRanges: { start: Date, end: Date }[] = [];

    // Si dateActive tiene más elementos que dateInactive, está activo hasta ahora
    if (dateActive.length > dateInactive.length) {
      // Crear rangos para todos los pares completos
      for (let i = 0; i < dateInactive.length; i++) {
        activeRanges.push({
          start: this.convertToUTCWithTimezone(dateActive[i]),
          end: this.convertToUTCWithTimezone(dateInactive[i])
        });
      }
      // El último rango activo va desde la última fecha de active hasta ahora
      activeRanges.push({
        start: this.convertToUTCWithTimezone(dateActive[dateActive.length - 1]),
        end: this.convertToUTCWithTimezone(now)
      });
    } else {
      // Si tienen la misma cantidad, crear rangos de fechas
      for (let i = 0; i < dateActive.length; i++) {
        activeRanges.push({
          start: this.convertToUTCWithTimezone(dateActive[i]),
          end: this.convertToUTCWithTimezone(dateInactive[i])
        });
      }
    }

    // Verificar si el plugin estaba activo en la fecha/hora exacta del trade
    for (const range of activeRanges) {
      if (tradeDateUTC >= range.start && tradeDateUTC <= range.end) {
        return range; // Plugin estaba activo en este rango
      }
    }

    return null; // Plugin no estaba activo
  }

  /**
   * PASO 2: Buscar la estrategia activa en la fecha/hora exacta del trade
```

```
   * MEJORA: Usar conversión UTC para comparaciones precisas
   * @param tradeDate - Fecha y hora exacta del trade
   * @returns nombre de la estrategia activa o null si no había ninguna
   */
  private getActiveStrategyAtTime(tradeDate: Date): string | null {
    if (!this.strategies || this.strategies.length === 0) {
      return null;
    }

    // MEJORA: Convertir fecha del trade a UTC para comparación precisa
    const tradeDateUTC = this.convertToUTCWithTimezone(tradeDate);


    // Buscar estrategias activas en la fecha/hora exacta del trade
    for (const strategy of this.strategies) {
      // IMPORTANTE: NO filtrar estrategias eliminadas aquí
      // Las estrategias eliminadas (soft delete) SÍ deben considerarse
      // porque en el momento del trade existían y podrían haber sido seguidas

      if (this.isStrategyActiveAtTime(strategy, tradeDateUTC)) {
        return strategy.name || 'Unknown Strategy';
      }
    }

    return null; // No había estrategia activa en ese momento
  }

  /**
   * Verificar si una estrategia específica estaba activa en la fecha/hora exacta
   * MEJORA: Usar conversión UTC para comparaciones precisas
   * @param strategy - Estrategia a verificar
   * @param tradeDate - Fecha y hora exacta del trade (ya en UTC)
   * @returns true si la estrategia estaba activa, false si no
   */
  private isStrategyActiveAtTime(strategy: ConfigurationOverview, tradeDate: Date): boolean {
    // Si la estrategia no tiene fechas de activación, no estaba activa
    if (!strategy.dateActive || !strategy.dateInactive) {
      return false;
    }

    const strategyActive = strategy.dateActive;
    const strategyInactive = strategy.dateInactive;
    const now = new Date();

    // Crear rangos de actividad de la estrategia
    const strategyRanges: { start: Date, end: Date }[] = [];

    // Si strategyActive tiene más elementos que strategyInactive, está activa hasta ahora
    if (strategyActive.length > strategyInactive.length) {
      // Crear rangos para todos los pares completos
      for (let i = 0; i < strategyInactive.length; i++) {
        strategyRanges.push({
          start:
this.convertToUTCWithTimezone(this.convertFirestoreTimestamp(strategyActive[i])),
          end:
this.convertToUTCWithTimezone(this.convertFirestoreTimestamp(strategyInactive[i]))
        });
      }
      // El último rango activo va desde la última fecha de active hasta ahora
      strategyRanges.push({
        start: this.convertToUTCWithTimezone(this.convertFirestoreTimestamp(strategyActive[st
rategyActive.length - 1])),
        end: this.convertToUTCWithTimezone(now)
      });
    } else {
      // Si tienen la misma cantidad, crear rangos de fechas
      for (let i = 0; i < strategyActive.length; i++) {
        strategyRanges.push({
          start:
this.convertToUTCWithTimezone(this.convertFirestoreTimestamp(strategyActive[i])),
          end:
this.convertToUTCWithTimezone(this.convertFirestoreTimestamp(strategyInactive[i]))
        });
      }
    }
```

```
431
432       // Verificar si la estrategia estaba activa en la fecha/hora exacta del trade
433       for (const range of strategyRanges) {
434         if (tradeDate >= range.start && tradeDate <= range.end) {
435           return true; // Estrategia estaba activa en este rango
436         }
437       }
438
439       return false; // Estrategia no estaba activa
440     }
441
442     /**
443      * Determinar si se siguió la estrategia basándose en los rangos de fechas del plugin
444      * @param tradeDate - Fecha del trade a validar
445      * @returns true si se siguió la estrategia, false si no
446      */
447     didFollowStrategy(tradeDate: Date): boolean {
448       return this.getStrategyFollowedOnDate(tradeDate) !== null;
449     }
450
451     /**
452      * Obtener información detallada sobre la estrategia seguida en un trade específico
453      * @param tradeDate - Fecha y hora exacta del trade
454      * @returns objeto con información detallada sobre la estrategia seguida
455      */
456     getTradeStrategyInfo(tradeDate: Date): {
457       followedStrategy: boolean;
458       strategyName: string | null;
459       pluginActive: boolean;
460       pluginActiveRange: { start: Date, end: Date } | null;
461       strategyActiveRange: { start: Date, end: Date } | null;
462     } {
463       const pluginActiveRange = this.getPluginActiveRange(tradeDate);
464       const strategyName = this.getActiveStrategyAtTime(tradeDate);
465
466       // Obtener el rango activo de la estrategia si existe
467       let strategyActiveRange: { start: Date, end: Date } | null = null;
468       if (strategyName && this.strategies) {
469         const strategy = this.strategies.find(s => s.name === strategyName);
470         if (strategy && strategy.dateActive && strategy.dateInactive) {
471           strategyActiveRange = this.getStrategyActiveRange(strategy, tradeDate);
472         }
473       }
474
475       return {
476         followedStrategy: strategyName !== null,
477         strategyName,
478         pluginActive: pluginActiveRange !== null,
479         pluginActiveRange,
480         strategyActiveRange
481       };
482     }
483
484     /**
485      * Obtener el rango activo de una estrategia específica en una fecha
486      * @param strategy - Estrategia a verificar
487      * @param tradeDate - Fecha del trade
488      * @returns rango activo de la estrategia o null si no estaba activa
489      */
490     private getStrategyActiveRange(strategy: ConfigurationOverview, tradeDate: Date): { start:
491   Date, end: Date } | null {
492       if (!strategy.dateActive || !strategy.dateInactive) {
493         return null;
494       }
495
496       const strategyActive = strategy.dateActive;
497       const strategyInactive = strategy.dateInactive;
498       const now = new Date();
499
500       // Crear rangos de actividad de la estrategia
       const strategyRanges: { start: Date, end: Date }[] = [];
```

```
501
502       // Si strategyActive tiene más elementos que strategyInactive, está activa hasta ahora
503       if (strategyActive.length > strategyInactive.length) {
504         // Crear rangos para todos los pares completos
505         for (let i = 0; i < strategyInactive.length; i++) {
506           strategyRanges.push({
507             start: this.convertFirestoreTimestamp(strategyActive[i]),
508             end: this.convertFirestoreTimestamp(strategyInactive[i])
509           });
510         }
511         // El último rango activo va desde la última fecha de active hasta ahora
512         strategyRanges.push({
513           start: this.convertFirestoreTimestamp(strategyActive[strategyActive.length - 1]),
514           end: now
515         });
516       } else {
517         // Si tienen la misma cantidad, crear rangos de fechas
518         for (let i = 0; i < strategyActive.length; i++) {
519           strategyRanges.push({
520             start: this.convertFirestoreTimestamp(strategyActive[i]),
521             end: this.convertFirestoreTimestamp(strategyInactive[i])
522           });
523         }
524       }
525
526       // Buscar el rango que contiene la fecha del trade
527       for (const range of strategyRanges) {
528         if (tradeDate >= range.start && tradeDate <= range.end) {
529           return range;
530         }
531       }
532
533       return null;
534     }
535
536     generateCalendar(targetMonth: Date) {
537       const tradesByDay: { [date: string]: GroupedTradeFinal[] } = {};
538
539       // Primero, filtrar trades válidos (con positionId válido) y deduplicar
540       const validTrades = this.groupedTrades.filter(trade =>
541         trade.positionId &&
542         trade.positionId !== 'null' &&
543         trade.positionId !== '' &&
544         trade.positionId !== null
545       );
546
547       // Deduplicar por positionId
548       const uniqueTrades = validTrades.filter((trade, index, self) =>
549         index === self.findIndex(t => t.positionId === trade.positionId)
550       );
551
552       // Agrupar trades únicos por día usando la zona horaria del dispositivo
553       uniqueTrades.forEach((trade) => {
554         // MEJORA: Usar conversión correcta de fecha de trade
555         const tradeDate = this.convertToUTCWithTimezone(Number(trade.lastModified));
556
557         // Usar la zona horaria local del dispositivo
558         const key = `${tradeDate.getFullYear()}-${tradeDate.getMonth()}-${tradeDate.getDate()}
559 `;
560         // Solo incluir trades que estén en el mes seleccionado
561         const tradeYear = tradeDate.getFullYear();
562         const tradeMonth = tradeDate.getMonth();
563         const targetYear = targetMonth.getFullYear();
564         const targetMonthIndex = targetMonth.getMonth();
565
566         if (tradeYear === targetYear && tradeMonth === targetMonthIndex) {
567           if (!tradesByDay[key]) tradesByDay[key] = [];
568           tradesByDay[key].push(trade);
569         }
570       });
```

```
571
572     // Generar calendario del mes objetivo
573     const year = targetMonth.getFullYear();
574     const month = targetMonth.getMonth();
575     const firstDay = new Date(year, month, 1);
576     const lastDay = new Date(year, month + 1, 0);
577
578     // Calcular inicio y fin de la semana
579     let startDay = new Date(firstDay);
580     startDay.setDate(firstDay.getDate() - firstDay.getDay());
581     let endDay = new Date(lastDay);
582     endDay.setDate(lastDay.getDate() + (6 - lastDay.getDay()));
583
584     const days: CalendarDay[] = [];
585     let currentDay = new Date(startDay);
586
587     while (currentDay <= endDay) {
588       const key = `${currentDay.getFullYear()}-${currentDay.getMonth()}-
589 ${currentDay.getDate()}`;
590       const trades = tradesByDay[key] || [];
591       const pnlTotal = trades.reduce((acc, t) => acc + (t.pnl ?? 0), 0);
592
593       const wins = trades.filter((t) => (t.pnl ?? 0) > 0).length;
594       const losses = trades.filter((t) => (t.pnl ?? 0) < 0).length;
595       const tradesCount = trades.length;
596       const tradeWinPercent = tradesCount > 0 ? Math.round((wins / tradesCount) * 1000) /
597 10 : 0;
598       // Determinar si se siguió la estrategia basándose en los rangos de fechas del plugin
599       // Para cada día, verificar si ALGÚN trade siguió la estrategia
600       let followedStrategy = false;
601       let strategyName: string | null = null;
602
603       if (tradesCount > 0) {
604         // Verificar cada trade individualmente usando su fecha/hora exacta
605         for (const trade of trades) {
606           // MEJORA: Usar conversión correcta de fecha de trade
607           const tradeDate = this.convertToUTCWithTimezone(Number(trade.lastModified));
608           const tradeStrategyInfo = this.getTradeStrategyInfo(tradeDate);
609
610           if (tradeStrategyInfo.followedStrategy) {
611             followedStrategy = true;
612             strategyName = tradeStrategyInfo.strategyName;
613             break; // Si al menos un trade siguió la estrategia, el día cuenta
614           }
615         }
616       }
617
618       days.push({
619         date: new Date(currentDay),
620         trades: trades as GroupedTradeFinal[],
621         pnlTotal,
622         tradesCount: trades.length,
623         followedStrategy: followedStrategy,
624         tradeWinPercent: Math.round(tradeWinPercent),
625         strategyName: strategyName,
626         isCurrentMonth: currentDay.getMonth() === month && currentDay.getFullYear() === year,
627       });
628
629       currentDay.setDate(currentDay.getDate() + 1);
630     }
631
632     // Organizar en semanas
633     this.calendar = [];
634     for (let i = 0; i < days.length; i += 7) {
635       this.calendar.push(days.slice(i, i + 7));
636     }
637
638   }
639
640   getDateNDaysAgo(daysAgo: number): Date {
     const date = new Date();
```

```
    date.setDate(date.getDate() - daysAgo);
    return date;
  }

  filterDaysInRange(
    days: CalendarDay[],
    fromDate: Date,
    toDate: Date
  ): CalendarDay[] {
    return days.filter((day) => day.date >= fromDate && day.date <= toDate);
  }

  countStrategyFollowedDays(days: CalendarDay[]): number {
    return days.filter((day) => day.followedStrategy && day.tradesCount > 0)
      .length;
  }

  calculateStrategyFollowedPercentage(
    days: CalendarDay[],
    periodDays: number
  ): number {
    if (days.length === 0) return 0;

    const fromDate = this.getDateNDaysAgo(periodDays - 1);
    const toDate = new Date();

    const daysInRange = this.filterDaysInRange(days, fromDate, toDate);
    const count = this.countStrategyFollowedDays(daysInRange);

    const percentage = (count / periodDays) * 100;

    return Math.round(percentage * 10) / 10;
  }

  getPercentageStrategyFollowedLast30Days() {
    const percentage = this.calculateStrategyFollowedPercentage(
      this.calendar.flat(),
      30
    );
    this.emitStrategyFollowedPercentage(percentage);
  }

  get currentMonthYear(): string {
    const options: Intl.DateTimeFormatOptions = { month: 'short' };
    const month = this.selectedMonth.toLocaleString('en-US', options);
    const year = this.selectedMonth.getFullYear();
    return `${month}, ${year}`;
  }

  // Navigation methods
  canNavigateLeft(): boolean {
    if (!this.groupedTrades || this.groupedTrades.length === 0) return false;

    const earliestTradeDate = this.getEarliestTradeDate();
    const firstDayOfSelectedMonth = new Date(this.selectedMonth.getFullYear(),
this.selectedMonth.getMonth(), 1);
    return earliestTradeDate < firstDayOfSelectedMonth;
  }

  canNavigateRight(): boolean {
    if (!this.groupedTrades || this.groupedTrades.length === 0) return false;

    const latestTradeDate = this.getLatestTradeDate();
    const lastDayOfSelectedMonth = new Date(this.selectedMonth.getFullYear(),
this.selectedMonth.getMonth() + 1, 0);
    return latestTradeDate > lastDayOfSelectedMonth;
  }

  private getEarliestTradeDate(): Date {
    if (!this.groupedTrades || this.groupedTrades.length === 0) return new Date();
```

```
    const dates = this.groupedTrades.map(trade => new Date(Number(trade.lastModified)));
    return new Date(Math.min(...dates.map(d => d.getTime())));
  }

  private getLatestTradeDate(): Date {
    if (!this.groupedTrades || this.groupedTrades.length === 0) return new Date();

    const dates = this.groupedTrades.map(trade => new Date(Number(trade.lastModified)));
    return new Date(Math.max(...dates.map(d => d.getTime())));
  }

  navigateToPreviousMonth(): void {
    if (this.canNavigateLeft()) {
      this.selectedMonth = new Date(this.selectedMonth.getFullYear(),
this.selectedMonth.getMonth() - 1);
      this.generateCalendar(this.selectedMonth);
    }
  }

  navigateToNextMonth(): void {
    if (this.canNavigateRight()) {
      this.selectedMonth = new Date(this.selectedMonth.getFullYear(),
this.selectedMonth.getMonth() + 1);
      this.generateCalendar(this.selectedMonth);
    }
  }

  navigateToCurrentMonth(): void {
    this.selectedMonth = new Date(this.currentDate.getFullYear(),
this.currentDate.getMonth());
    this.generateCalendar(this.selectedMonth);
  }

  // Export functionality
  exportData() {
    const csvData = this.generateCSVData();
    this.downloadCSV(csvData, `trading-data-${this.currentMonthYear.replace(', ',
'-').toLowerCase()}.csv`);
  }

  generateCSVData(): string {
    const headers = ['Date', 'PnL Total', 'Trades Count', 'Win Percentage', 'Strategy
Followed', 'Strategy Name'];
    const rows = [headers.join(',')];

    this.calendar.flat().forEach(day => {
      const date = day.date.toISOString().split('T')[0];
      const pnlTotal = day.pnlTotal.toFixed(2);
      const tradesCount = day.tradesCount;
      const winPercentage = day.tradeWinPercent;
      const strategyFollowed = day.followedStrategy ? 'Yes' : 'No';
      const strategyName = day.strategyName || 'None';

      rows.push([date, pnlTotal, tradesCount, winPercentage, strategyFollowed,
strategyName].join(','));
    });

    return rows.join('\n');
  }

  downloadCSV(csvData: string, filename: string) {
    const blob = new Blob([csvData], { type: 'text/csv;charset=utf-8;' });
    const link = document.createElement('a');

    if (link.download !== undefined) {
      const url = URL.createObjectURL(blob);
      link.setAttribute('href', url);
      link.setAttribute('download', filename);
      link.style.visibility = 'hidden';
      document.body.appendChild(link);
      link.click();
      document.body.removeChild(link);
    }
  }
```

```
  // Weekly summary methods
  getWeekTotal(week: CalendarDay[]): number {
    return week.reduce((total, day) => total + day.pnlTotal, 0);
  }

  getWeekActiveDays(week: CalendarDay[]): number {
    return week.filter(day => day.tradesCount > 0).length;
  }

  // Popup methods
  onDayClick(day: CalendarDay) {
    if (day.tradesCount > 0) {
      this.selectedDay = day;
      this.showTradesPopup = true;
    }
  }

  onClosePopup() {
    this.showTradesPopup = false;
    this.selectedDay = null;
  }

  formatCurrency(value: number): string {
    return this.numberFormatter.formatCurrency(value);
  }

  formatPercentage(value: number): string {
    return this.numberFormatter.formatPercentage(value);
  }

  /**
   * Obtener resumen de estrategias seguidas en un período
   * @param days - Array de días del calendario
   * @returns objeto con estadísticas de estrategias
   */
  getStrategySummary(days: CalendarDay[]): {
    totalDays: number;
    strategyDays: number;
    strategiesUsed: { [strategyName: string]: number };
    strategyPercentage: number;
  } {
    const totalDays = days.length;
    let strategyDays = 0;
    const strategiesUsed: { [strategyName: string]: number } = {};

    days.forEach(day => {
      if (day.followedStrategy && day.strategyName) {
        strategyDays++;
        if (strategiesUsed[day.strategyName]) {
          strategiesUsed[day.strategyName]++;
        } else {
          strategiesUsed[day.strategyName] = 1;
        }
      }
    });

    const strategyPercentage = totalDays > 0 ? Math.round((strategyDays / totalDays) * 100 *
10) / 10 : 0;
    return {
      totalDays,
      strategyDays,
      strategiesUsed,
      strategyPercentage
    };
  }

  /**
   * Obtener resumen de estrategias para los últimos N días
   * @param days - Número de días a analizar
   * @returns resumen de estrategias
```

```
   */
  getStrategySummaryLastNDays(days: number): {
    totalDays: number;
    strategyDays: number;
    strategiesUsed: { [strategyName: string]: number };
    strategyPercentage: number;
  } {
    const fromDate = this.getDateNDaysAgo(days - 1);
    const toDate = new Date();
    const daysInRange = this.filterDaysInRange(this.calendar.flat(), fromDate, toDate);

    return this.getStrategySummary(daysInRange);
  }

  /**
   * Obtener análisis detallado de trades que siguieron estrategias
   * @param days - Array de días del calendario
   * @returns análisis detallado de trades y estrategias
   */
  getDetailedTradeAnalysis(days: CalendarDay[]): {
    totalTrades: number;
    tradesWithStrategy: number;
    tradesWithoutStrategy: number;
    strategyCompliance: number;
    tradesByStrategy: { [strategyName: string]: number };
    tradesWithoutPlugin: number;
    tradesWithPluginButNoStrategy: number;
  } {
    let totalTrades = 0;
    let tradesWithStrategy = 0;
    let tradesWithoutStrategy = 0;
    let tradesWithoutPlugin = 0;
    let tradesWithPluginButNoStrategy = 0;
    const tradesByStrategy: { [strategyName: string]: number } = {};

    days.forEach(day => {
      if (day.trades && day.trades.length > 0) {
        day.trades.forEach(trade => {
          totalTrades++;
          // MEJORA: Usar conversión correcta de fecha de trade
          const tradeDate = this.convertToUTCWithTimezone(Number(trade.lastModified));
          const tradeStrategyInfo = this.getTradeStrategyInfo(tradeDate);

          if (!tradeStrategyInfo.pluginActive) {
            tradesWithoutPlugin++;
          } else if (tradeStrategyInfo.followedStrategy) {
            tradesWithStrategy++;
            const strategyName = tradeStrategyInfo.strategyName || 'Unknown';
            tradesByStrategy[strategyName] = (tradesByStrategy[strategyName] || 0) + 1;
          } else {
            tradesWithPluginButNoStrategy++;
          }
        });
      }
    });

    tradesWithoutStrategy = tradesWithoutPlugin + tradesWithPluginButNoStrategy;
    const strategyCompliance = totalTrades > 0 ? Math.round((tradesWithStrategy /
totalTrades) * 100 * 10) / 10 : 0;
    return {
      totalTrades,
      tradesWithStrategy,
      tradesWithoutStrategy,
      strategyCompliance,
      tradesByStrategy,
      tradesWithoutPlugin,
      tradesWithPluginButNoStrategy
    };
  }

```

```
  /**
   * Obtener análisis detallado de trades para los últimos N días
   * @param days - Número de días a analizar
   * @returns análisis detallado de trades
   */
  getDetailedTradeAnalysisLastNDays(days: number): {
    totalTrades: number;
    tradesWithStrategy: number;
    tradesWithoutStrategy: number;
    strategyCompliance: number;
    tradesByStrategy: { [strategyName: string]: number };
    tradesWithoutPlugin: number;
    tradesWithPluginButNoStrategy: number;
  } {
    const fromDate = this.getDateNDaysAgo(days - 1);
    const toDate = new Date();
    const daysInRange = this.filterDaysInRange(this.calendar.flat(), fromDate, toDate);

    return this.getDetailedTradeAnalysis(daysInRange);
  }

  /**
   * Verificar estado actual del plugin
   * @returns true si el plugin está activo ahora, false si no
   */
  isPluginCurrentlyActive(): boolean {
    if (!this.pluginHistory) {
      return false;
    }

    return this.pluginHistoryService.isPluginActiveByDates(this.pluginHistory);
  }

  /**
   * Obtener información detallada del estado del plugin
   * @returns información completa del estado del plugin
   */
  getPluginStatusInfo(): {
    isActive: boolean;
    lastActiveDate: string | null;
    lastInactiveDate: string | null;
    activeRanges: { start: string, end: string }[];
  } {
    if (!this.pluginHistory) {
      return {
        isActive: false,
        lastActiveDate: null,
        lastInactiveDate: null,
        activeRanges: []
      };
    }

    const isActive = this.pluginHistoryService.isPluginActiveByDates(this.pluginHistory);
    const lastActiveDate = this.pluginHistory.dateActive?.
[this.pluginHistory.dateActive.length - 1] || null;
    const lastInactiveDate = this.pluginHistory.dateInactive?.
[this.pluginHistory.dateInactive.length - 1] || null;
    // Crear rangos activos para mostrar
    const activeRanges: { start: string, end: string }[] = [];
    if (this.pluginHistory.dateActive && this.pluginHistory.dateInactive) {
      const dateActive = this.pluginHistory.dateActive;
      const dateInactive = this.pluginHistory.dateInactive;

      for (let i = 0; i < Math.min(dateActive.length, dateInactive.length); i++) {
        activeRanges.push({
          start: new Date(dateActive[i]).toISOString(),
          end: new Date(dateInactive[i]).toISOString()
        });
      }
    }

```

```
    return {
      isActive,
      lastActiveDate,
      lastInactiveDate,
      activeRanges
    };
  }

}

```

Ø=ÜÁ features\report\components\pnlGraph

Ø=ÜÄ features\report\components\pnlGraph\pnlGraph.component.ts

```typescript
import { CommonModule } from '@angular/common';
import {
  Component,
  Inject,
  Input,
  OnInit,
  OnChanges,
  Output,
  PLATFORM_ID,
  EventEmitter,
  SimpleChanges,
  HostListener,
} from '@angular/core';
import { GroupedTrade, GroupedTradeFinal } from '../../models/report.model';
import { NgApexchartsModule } from 'ng-apexcharts';
import { getMonthlyPnL } from '../../utils/normalization-utils';
import { FormsModule } from '@angular/forms';
import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';

/**
 * Component for displaying PnL (Profit and Loss) chart.
 *
 * This component displays an area chart showing cumulative PnL over time.
 * It supports filtering by year or custom date range, and displays monthly
 * or dynamic date-based aggregations depending on the selected range.
 *
 * Features:
 * - Area chart visualization using ApexCharts
 * - Year-based filtering (past 2 years, current year, next 5 years)
 * - Custom date range filtering
 * - Monthly aggregation for year view
 * - Dynamic aggregation (daily/weekly/monthly) for date ranges
 * - Total profit calculation for filtered data
 * - Interactive tooltips with percentage changes
 *
 * Relations:
 * - NgApexchartsModule: Chart rendering
 * - NumberFormatterService: Value formatting
 *
 * @component
 * @selector app-PnL-Graph
 * @standalone true
 */
@Component({
  selector: 'app-PnL-Graph',
  templateUrl: './pnlGraph.component.html',
  styleUrls: ['./pnlGraph.component.scss'],
  standalone: true,
```

```
  imports: [CommonModule, NgApexchartsModule, FormsModule],
})
export class PnlGraphComponent implements OnInit, OnChanges {
  @Input() values!: GroupedTradeFinal[];
  @Output() onYearChange = new EventEmitter<string>();

  public chartOptions: any;
  private numberFormatter = new NumberFormatterService();

  year!: string;
  dateRanges: { label: string; value: string }[] = [];

  // Date filter properties
  showDateFilter = false;
  selectedStartDate: string = '';
  selectedEndDate: string = '';
  filteredData: GroupedTradeFinal[] = [];
  originalData: GroupedTradeFinal[] = [];

  constructor(@Inject(PLATFORM_ID) private platformId: any) {}

  ngOnInit() {
    this.year = new Date().getFullYear().toString();
    this.generateYearRangesPast(3);
    this.initializeData();
  }

  ngOnChanges(changes: SimpleChanges) {
    if (changes['values'] && this.values) {
      this.initializeData();
    }
  }

  private initializeData() {
    if (this.values && this.values.length > 0) {
      this.originalData = [...this.values];
      this.filteredData = [...this.values];
      this.chartOptions = this.getChartOptions(this.filteredData);
    } else {
      // Mostrar gráfica vacía cuando no hay datos
      this.originalData = [];
      this.filteredData = [];
      this.chartOptions = this.getEmptyChartOptions();
    }
  }

  generateYearRangesPast(yearsBack: number) {
    const now = new Date();
    const currentYear = now.getFullYear();
    this.dateRanges = [];

    // Generate 2 years before current and 5 years after
    for (let i = 2; i >= 0; i--) {
      const year = currentYear - i;
      this.dateRanges.push({
        label: `Jan ${year} - Dec ${year}`,
        value: `${year}`,
      });
    }

    for (let i = 1; i <= 5; i++) {
      const year = currentYear + i;
      this.dateRanges.push({
        label: `Jan ${year} - Dec ${year}`,
        value: `${year}`,
      });
    }
  }

  getChartOptions(trades: GroupedTradeFinal[]): any {
```

```
    const yearValue = this.year;

    // Si hay filtro de fechas activo, no aplicar filtro de año
    let filteredTrades = trades;
    if (!this.selectedStartDate && !this.selectedEndDate) {
      filteredTrades = this.applyYearFilter(trades);
    }

    // Use monthly chart by default
    const chartConfig = this.getMonthlyChartConfig(filteredTrades, yearValue);

    return {
      chart: {
        type: 'area',
        height: 350,
        toolbar: { show: false },
        foreColor: '#fff',
        fontFamily: 'Inter, Arial, sans-serif',
        background: 'transparent',
      },
      series: [
        {
          name: 'PnL',
          data: chartConfig.data,
        },
      ],
      xaxis: {
        categories: chartConfig.categories,
        labels: {
          style: { colors: '#d8d8d8' },
        },
        axisBorder: { show: false },
        axisTicks: { show: false },
      },
      yaxis: {
        labels: {
          style: { colors: '#d8d8d8' },
        },
      },
      grid: {
        borderColor: '#333',
        strokeDashArray: 4,
        xaxis: {
          lines: {
            show: true,
          },
        },
      },
      dataLabels: { enabled: false },
      stroke: {
        curve: 'straight',
        width: 1,
        colors: ['#EAF2F8'],
      },
      fill: {
        type: 'gradient',
        gradient: {
          shade: 'dark',
          type: 'vertical',
          gradientToColors: ['#3967D7'],
          opacityFrom: 0.4,
          opacityTo: 0,
        },
      },
      tooltip: {
        theme: 'dark',
        x: { show: true },
        custom: ({ series, seriesIndex, dataPointIndex, w }: any) => {
          const value = series[seriesIndex][dataPointIndex];
          const category = w.globals.categoryLabels[dataPointIndex];
```

```
          const prevValue =
            dataPointIndex > 0 ? series[seriesIndex][dataPointIndex - 1] : null;
          let percentDiff: number | null = 0;
          let direction = null;
          let cardClass = '';
          let validatorClass = '';

          if (prevValue !== null && prevValue !== 0) {
            percentDiff = ((value - prevValue) / Math.abs(prevValue)) * 100;
            direction = percentDiff > 0 ? 'up' : 'down';
            if (direction === 'up') {
              cardClass = 'positive-container';
              validatorClass = 'positive-validator';
            } else {
              cardClass = 'negative-container';
              validatorClass = 'negative-validator';
            }
          } else {
            percentDiff = null;
            direction = null;
          }

          const formattedValue = this.getFormatedValue(value);
          const formattedPercent = this.numberFormatter.formatPercentageValue(percentDiff);

          return `<div class=" ${cardClass} regularText color-background d-flex flex-col
toolTip-container items-start">
          <p class="smallText color-text-gray">${category}, ${yearValue}</p>
          <div class="d-flex text-container items-center ">
                    <p class= "subtitle">
                    ${formattedValue}
          </p>
          ${
            percentDiff != null
              ? `<span class="smallText py-4 px-6 d-flex justify-center items-center
${validatorClass}">${formattedPercent}% <span class="${
                    direction === 'up'
                    ? 'icon-status-arrow-up'
                    : 'icon-status-arrow'
                } ml-3"></span></span>`
              : ''
          }
          </div>


          </div>`;
        },
        position: function (data: any, opts: any) {
          return {
            left: data.point.x,
            top: data.point.y - 160,
          };
        },
      },
    };
  }

  applyYearFilter(trades: GroupedTradeFinal[]): GroupedTradeFinal[] {
    if (!trades || trades.length === 0) return [];

    let filteredTrades = [...trades];

    // Filter by year
    const yearValue = parseInt(this.year);
    filteredTrades = filteredTrades.filter(trade => {
      const tradeDate = new Date(Number(trade.lastModified));
      return tradeDate.getFullYear() === yearValue;
    });

    return filteredTrades;
  }
```

```
259
260   applyDateRangeFilter(trades: GroupedTradeFinal[], startDate: string, endDate: string):
261 GroupedTradeFinal[] {
262     if (!startDate && !endDate) {
263       return trades;
264     }
265
266     const start = startDate ? new Date(startDate) : new Date(0);
267     const end = endDate ? new Date(endDate) : new Date();
268
269     // Si solo hay fecha de inicio, usar fin del día
270     if (startDate && !endDate) {
271       end.setHours(23, 59, 59, 999);
272     }
273     // Si solo hay fecha de fin, usar inicio del día
274     if (!startDate && endDate) {
275       start.setHours(0, 0, 0, 0);
276     }
277
278     return trades.filter(trade => {
279       const tradeDate = new Date(Number(trade.lastModified));
280       return tradeDate >= start && tradeDate <= end;
281     });
282   }
283
284
285   getMonthlyChartConfig(trades: GroupedTradeFinal[], yearValue: string) {
286     const monthlyMap: { [label: string]: number } = {};
287
288     // Si hay filtro de fechas activo, generar categorías dinámicas
289     if (this.selectedStartDate || this.selectedEndDate) {
290       return this.getDateRangeChartConfig(trades);
291     }
292
293     trades.forEach(trade => {
294       const date = new Date(Number(trade.lastModified));
295       const tradeYear = date.getFullYear();
296       const yearValueNum = parseInt(yearValue);
297
298       // Only process trades from the selected year
299       if (tradeYear === yearValueNum) {
300         const label = this.capitalizeFirstLetter(
301           date.toLocaleString('en', { month: 'short' })
302         );
303         const sum = (monthlyMap[label] ?? 0) + (trade.pnl ?? 0);
304         monthlyMap[label] = sum < 1 ? Math.round(sum * 100) / 100 : Math.round(sum);
305
306       }
307     });
308
309     const monthOrder = [
310       'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
311       'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',
312     ];
313
314     const data = monthOrder.map(m => monthlyMap[m] ?? 0);
315     const categories = monthOrder;
316     return { data, categories };
317   }
318
319   getDateRangeChartConfig(trades: GroupedTradeFinal[]) {
320     const dateMap: { [key: string]: number } = {};
321
322     // Determinar el rango de fechas
323     let startDate: Date;
324     let endDate: Date;
325
326     if (this.selectedStartDate && this.selectedEndDate) {
327       startDate = new Date(this.selectedStartDate);
328       endDate = new Date(this.selectedEndDate);
     } else if (this.selectedStartDate) {
```

```
      startDate = new Date(this.selectedStartDate);
      endDate = new Date();
    } else if (this.selectedEndDate) {
      startDate = new Date(0);
      endDate = new Date(this.selectedEndDate);
    } else {
      // Fallback a año completo
      const year = new Date().getFullYear();
      startDate = new Date(year, 0, 1);
      endDate = new Date(year, 11, 31);
    }

    // Procesar trades en el rango
    trades.forEach(trade => {
      const tradeDate = new Date(Number(trade.lastModified));
      if (tradeDate >= startDate && tradeDate <= endDate) {
        const label = this.formatDateForChart(tradeDate);
        const sum = (dateMap[label] ?? 0) + (trade.pnl ?? 0);
        dateMap[label] = sum < 1 ? Math.round(sum * 100) / 100 : Math.round(sum);
      }
    });

    // Generar categorías dinámicas basadas en el rango
    const categories = this.generateDateRangeCategories(startDate, endDate);
    const data = categories.map(cat => dateMap[cat] ?? 0);

    return { data, categories };
  }

  formatDateForChart(date: Date): string {
    const month = date.toLocaleString('en', { month: 'short' });
    const day = date.getDate();
    return `${month} ${day}`;
  }

  generateDateRangeCategories(startDate: Date, endDate: Date): string[] {
    const categories: string[] = [];
    const current = new Date(startDate);

    // Si el rango es menor a 30 días, mostrar por días
    const diffTime = endDate.getTime() - startDate.getTime();
    const diffDays = Math.ceil(diffTime / (1000 * 60 * 60 * 24));

    if (diffDays <= 30) {
      // Mostrar por días
      while (current <= endDate) {
        categories.push(this.formatDateForChart(new Date(current)));
        current.setDate(current.getDate() + 1);
      }
    } else if (diffDays <= 90) {
      // Mostrar por semanas
      while (current <= endDate) {
        const weekEnd = new Date(current);
        weekEnd.setDate(weekEnd.getDate() + 6);
        if (weekEnd > endDate) weekEnd.setTime(endDate.getTime());

        const startStr = this.formatDateForChart(new Date(current));
        const endStr = this.formatDateForChart(weekEnd);
        categories.push(`${startStr} - ${endStr}`);

        current.setDate(current.getDate() + 7);
      }
    } else {
      // Mostrar por meses
      while (current <= endDate) {
        const month = current.toLocaleString('en', { month: 'short' });
        const year = current.getFullYear();
        categories.push(`${month} ${year}`);

        current.setMonth(current.getMonth() + 1);
```

```
      }
    }

    return categories;
  }

  capitalizeFirstLetter(str: string): string {
    if (!str) return '';
    return str.charAt(0).toUpperCase() + str.slice(1);
  }

  onYearSelected(year: string) {
    this.year = year;
    this.onYearChange.emit(year);
    this.updateChart();
  }

  get getTotalProfit(): number {
    // Apply the same filters as the chart
    let filteredTrades = this.filteredData;
    if (!this.selectedStartDate && !this.selectedEndDate) {
      filteredTrades = this.applyYearFilter(this.filteredData);
    }

    const totalProfit = filteredTrades.reduce(
      (acc, trade) => acc + (trade.pnl ?? 0),
      0
    );
    const result = Math.round(totalProfit * 100) / 100;
    return result;
  }

  // Date filter methods
  toggleDateFilter() {
    this.showDateFilter = !this.showDateFilter;
  }

  closeDateFilter() {
    this.showDateFilter = false;
  }

  @HostListener('document:click', ['$event'])
  onDocumentClick(event: Event) {
    const target = event.target as HTMLElement;
    const filterSection = target.closest('.filter-section');
    if (!filterSection && this.showDateFilter) {
      this.closeDateFilter();
    }
  }

  onStartDateSelected(date: string) {
    this.selectedStartDate = date;
  }

  onEndDateSelected(date: string) {
    this.selectedEndDate = date;
  }

  applyDateFilter() {
    if (this.selectedStartDate || this.selectedEndDate) {
      this.filteredData = this.applyDateRangeFilter(this.originalData,
this.selectedStartDate, this.selectedEndDate);
      // Actualizar el año basado en las fechas seleccionadas
      this.updateYearFromDateRange();
    } else {
      this.filteredData = [...this.originalData];
    }
    this.updateChart();
    this.closeDateFilter();
  }

```

```
  updateYearFromDateRange() {
    if (this.selectedStartDate) {
      const startDate = new Date(this.selectedStartDate);
      this.year = startDate.getFullYear().toString();
    } else if (this.selectedEndDate) {
      const endDate = new Date(this.selectedEndDate);
      this.year = endDate.getFullYear().toString();
    }
  }

  clearDateFilter() {
    this.selectedStartDate = '';
    this.selectedEndDate = '';
    this.filteredData = [...this.originalData];
    this.updateChart();
  }

  updateChart() {
    this.chartOptions = this.getChartOptions(this.filteredData);
  }

  getAvailableYears(): string[] {
    if (!this.values || this.values.length === 0) return [];

    const years = new Set<number>();
    this.values.forEach(trade => {
      const date = new Date(Number(trade.lastModified));
      years.add(date.getFullYear());
    });

    return Array.from(years).sort((a, b) => b - a).map(year => year.toString());
  }

  getEmptyChartOptions(): any {
    const currentYear = new Date().getFullYear();
    const months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
'Nov', 'Dec'];
    return {
      chart: {
        type: 'area',
        height: 350,
        toolbar: { show: false },
        foreColor: '#fff',
        fontFamily: 'Inter, Arial, sans-serif',
        background: 'transparent',
      },
      series: [
        {
          name: 'PnL',
          data: new Array(12).fill(0), // Array de 12 ceros para los 12 meses
        },
      ],
      xaxis: {
        categories: months,
        labels: {
          style: { colors: '#d8d8d8' },
        },
        axisBorder: { show: false },
        axisTicks: { show: false },
      },
      yaxis: {
        labels: {
          style: { colors: '#d8d8d8' },
        },
      },
      grid: {
        borderColor: '#333',
        strokeDashArray: 4,
        xaxis: {
          lines: {
```

```
            show: true,
          },
        },
      },
      dataLabels: { enabled: false },
      stroke: {
        curve: 'straight',
        width: 1,
        colors: ['#EAF2F8'],
      },
      fill: {
        type: 'gradient',
        gradient: {
          shade: 'dark',
          type: 'vertical',
          gradientToColors: ['#3967D7'],
          opacityFrom: 0.4,
          opacityTo: 0,
        },
      },
      tooltip: {
        theme: 'dark',
        x: { show: true },
        custom: function ({ series, seriesIndex, dataPointIndex, w }: any) {
          const month = months[dataPointIndex];
          return `
            <div style="padding: 8px 12px; background: #1a1a1a; border: 1px solid #333;
border-radius: 8px;">
              <div style="color: #d8d8d8; font-size: 12px;">${month} ${currentYear}</div>
              <div style="color: #fff; font-size: 14px; font-weight: 600;">PnL: $0.00</div>
            </div>
          `;
        },
      },
      noData: {
        text: 'No data available',
        align: 'center',
        verticalAlign: 'middle',
        style: {
          color: '#d8d8d8',
          fontSize: '14px',
          fontFamily: 'Inter, Arial, sans-serif'
        }
      }
    };
  }

  getFormatedValue(value: number): string {

    return this.numberFormatter.formatCurrencyValue(value);

  }
}
```

Ø=ÜÁ features\report\components\rule-short

Ø=ÜÄ features\report\components\rule-short\rule-short.component.ts

```
import { CommonModule } from '@angular/common';
import { Component, Input } from '@angular/core';

/**
 * Component for displaying a short rule indicator.
```

```
 *
 * This component displays a rule title and an active/inactive status indicator.
 * It is used in the report view to show which trading rules are currently active.
 *
 * @component
 * @selector app-rule-short
 * @standalone true
 */
@Component({
  selector: 'app-rule-short',
  templateUrl: './rule-short.component.html',
  styleUrls: ['./rule-short.component.scss'],
  standalone: true,
  imports: [CommonModule],
})
export class RuleShortComponent {
  @Input() title!: string;
  @Input() isActive?: any;

  constructor() {}
}

```

Ø=ÜÁ features\report\components\statCard

Ø=ÜÄ features\report\components\statCard\stat_card.component.ts

```
import { CommonModule } from '@angular/common';
import { Component, Input, Injectable } from '@angular/core';
import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';

/**
 * Component for displaying a statistical card with formatted values.
 *
 * This component displays a title and value with automatic formatting based on the format type
 * or by auto-detecting the format from the title. It also applies color coding for certain metrics.
 * Format types:
 * - 'currency': Formats as currency (e.g., $1,234.56)
 * - 'percentage': Formats as percentage (e.g., 45.5%)
 * - 'number': Formats as number or integer based on title
 *
 * Color coding:
 * - Net P&L: Green for positive, red for negative
 * - Profit Factor: Green if >= 1.0, red if < 1.0
 * - Trade Win %: Green if >= 50%, red if < 50%
 * - Avg Win/Loss: Green if >= 1.0, red if < 1.0
 *
 * @component
 * @selector app-stat-card
 * @standalone true
 */
@Component({
  selector: 'app-stat-card',
  templateUrl: './stat_card.component.html',
  styleUrls: ['./stat_card.component.scss'],
  standalone: true,
  imports: [CommonModule],
})
@Injectable()
export class statCardComponent {
  @Input() title!: string;
  @Input() value?: string | number;
```

```
  @Input() formatType?: 'currency' | 'percentage' | 'number';

  private numberFormatter = new NumberFormatterService();

  /**
   * Gets the formatted value based on format type or auto-detection.
   *
   * If formatType is specified, uses that format. Otherwise, auto-detects format
   * from the title (e.g., titles containing "P&L" or "profit" use currency format).
   *
   * Related to:
   * - NumberFormatterService: Handles actual formatting
   *
   * @returns Formatted value string
   * @memberof statCardComponent
   */
  getFormattedValue(): string {
    if (this.value === null || this.value === undefined) {
      return '0';
    }

    switch (this.formatType) {
      case 'currency':
        return this.numberFormatter.formatCurrency(this.value);
      case 'percentage':
        return this.numberFormatter.formatPercentage(this.value);
      case 'number':
        // Check if this is a count (like total trades, active positions) that should be an
integer if (this.title.toLowerCase().includes('total') &&
            this.title.toLowerCase().includes('trade')) {
          return this.numberFormatter.formatInteger(this.value);
        }
        if (this.title.toLowerCase().includes('active') &&
            this.title.toLowerCase().includes('position')) {
          return this.numberFormatter.formatInteger(this.value);
        }
        if (this.title.toLowerCase().includes('users')) {
          return this.numberFormatter.formatInteger(this.value);
        }
        if (this.title.toLowerCase().includes('subscriptions')) {
          return this.numberFormatter.formatInteger(this.value);
        }
        return this.numberFormatter.formatNumber(this.value);
      default:
        // Auto-detect format based on title
        if (this.title.toLowerCase().includes('p&l') ||
            this.title.toLowerCase().includes('revenue') ||
            this.title.toLowerCase().includes('sales') ||
            this.title.toLowerCase().includes('profit') ||
            this.title.toLowerCase().includes('balance')) {
          return this.numberFormatter.formatCurrency(this.value);
        } else if (this.title.toLowerCase().includes('%') ||
                   this.title.toLowerCase().includes('percent') ||
                   this.title.toLowerCase().includes('win rate')) {
          return this.numberFormatter.formatPercentage(this.value);
        } else if (this.title.toLowerCase().includes('total') &&
                   this.title.toLowerCase().includes('trade')) {
          return this.numberFormatter.formatInteger(this.value);
        } else if (this.title.toLowerCase().includes('active') &&
                   this.title.toLowerCase().includes('position')) {
          return this.numberFormatter.formatInteger(this.value);
        } else if (this.title.toLowerCase().includes('users')) {
          return this.numberFormatter.formatInteger(this.value);
        } else if (this.title.toLowerCase().includes('subscriptions')) {
          return this.numberFormatter.formatInteger(this.value);
        } else {
          return this.numberFormatter.formatNumber(this.value);
        }
    }
  }
```

```
107
108   /**
109    * Gets the CSS color class for the value based on the metric type and value.
110    *
111    * Applies color coding for specific metrics:
112    * - Net P&L: Green (positive) or red (negative)
113    * - Profit Factor: Green (>= 1.0) or red (< 1.0)
114    * - Trade Win %: Green (>= 50%) or red (< 50%)
115    * - Avg Win/Loss: Green (>= 1.0) or red (< 1.0)
116    * - Other metrics: Default background color
117    *
118    * @returns CSS class name for value color
119    * @memberof statCardComponent
120    */
121   getValueColorClass(): string {
122     if (this.value === null || this.value === undefined) {
123       return 'color-background';
124     }
125
126     const numericValue = Number(this.value);
127
128     // Solo aplicar colores a métricas específicas
129
130     // Para Net P&L: rojo si es negativo, verde si es positivo
131     if (this.title.toLowerCase().includes('p&l')) {
132       return numericValue < 0 ? 'color-error' : 'color-success';
133     }
134
135     // Para Profit Factor: rojo si es menor a 1.0, verde si es mayor a 1.0
136     if (this.title.toLowerCase().includes('profit') &&
137         this.title.toLowerCase().includes('factor')) {
138       return numericValue < 1.0 ? 'color-error' : 'color-success';
139     }
140
141     // Para Trade Win %: rojo si es menor a 50%, verde si es mayor a 50%
142     if (this.title.toLowerCase().includes('win') &&
143         this.title.toLowerCase().includes('%')) {
144       return numericValue < 50 ? 'color-error' : 'color-success';
145     }
146
147     // Para Avg Win/Loss: rojo si es menor a 1.0, verde si es mayor a 1.0
148     if (this.title.toLowerCase().includes('avg') &&
149         this.title.toLowerCase().includes('win')) {
150       return numericValue < 1.0 ? 'color-error' : 'color-success';
151     }
152
153     // Para Balance, Total trades, Active positions: siempre blanco
154     return 'color-background';
155   }
156 }
157
```

Ø=ÜÁ features\report\components\trades-popup

Ø=ÜÄ features\report\components\trades-popup\trades-popup.component.ts

```
1 import { CommonModule } from '@angular/common';
2 import { Component, Input, Output, EventEmitter } from '@angular/core';
3 import { CalendarDay } from '../../models/report.model';
4 import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';
5 import { GroupedTradeFinal } from '../../models/report.model';
6 import { ConfigurationOverview } from '../../../strategy/models/strategy.model';
7
```

```typescript
/**
 * Interface representing a trade detail for display in the popup.
 *
 * @interface TradeDetail
 */
export interface TradeDetail {
  openTime: string;
  ticker: string;
  side: 'Long' | 'Short';
  netPnl: number;
  followedStrategy: boolean;
  strategyName: string;
}

/**
 * Component for displaying trades in a popup modal.
 *
 * This component displays detailed information about trades for a selected day,
 * including trade time, ticker, side (Long/Short), PnL, and strategy compliance.
 *
 * Features:
 * - Displays all trades for a selected calendar day
 * - Shows trade details: time, ticker, side, PnL
 * - Indicates if trades followed a strategy
 * - Color-coded tickers and PnL values
 * - Formatted currency and percentage values
 *
 * Relations:
 * - CalendarComponent: Receives selected day data
 * - NumberFormatterService: Value formatting
 *
 * @component
 * @selector app-trades-popup
 * @standalone true
 */
@Component({
  selector: 'app-trades-popup',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './trades-popup.component.html',
  styleUrls: ['./trades-popup.component.scss']
})
export class TradesPopupComponent {
  @Input() visible: boolean = false;
  @Input() selectedDay: CalendarDay | null = null;
  @Input() strategies: ConfigurationOverview[] = [];
  @Output() close = new EventEmitter<void>();

  trades: TradeDetail[] = [];
  netPnl: number = 0;
  netRoi: number = 0;
  private numberFormatter = new NumberFormatterService();
  selectedDate: string = '';

  // Expose Math to template
  Math = Math;

  ngOnChanges() {
    if (this.selectedDay && this.visible) {
      this.loadTradesData();
    }
  }

  loadTradesData() {
    if (!this.selectedDay) return;

    this.selectedDate = this.formatDate(this.selectedDay.date);
    this.netPnl = this.selectedDay.pnlTotal;

    console.log(this.selectedDay.trades);
```

```
78
79      // Convertir trades del día a formato de detalle
80      this.trades = this.selectedDay.trades.map((trade, index) => ({
81        openTime: this.formatTime(new Date(Number(trade.lastModified))),
82        ticker: trade.instrument ?? 'N/A',
83        side: this.determineSide(trade),
84        netPnl: trade.pnl ?? 0,
85        followedStrategy: this.selectedDay?.followedStrategy ?? false,
86        strategyName: this.getStrategyNameForTrade(trade)
87      }));
88
89      // Ordenar por tiempo (más reciente primero)
90      this.trades.sort((a, b) => b.openTime.localeCompare(a.openTime));
91    }
92
93    formatDate(date: Date): string {
94      const options: Intl.DateTimeFormatOptions = {
95        weekday: 'long',
96        year: 'numeric',
97        month: 'long',
98        day: 'numeric'
99      };
100     return date.toLocaleDateString('en-US', options);
101   }
102
103   formatTime(date: Date): string {
104     return date.toLocaleTimeString('en-US', {
105       hour12: false,
106       hour: '2-digit',
107       minute: '2-digit',
108       second: '2-digit'
109     });
110   }
111
112   determineSide(trade: GroupedTradeFinal): 'Long' | 'Short' {
113     // Usar el campo side real del trade para determinar Long/Short
114     if (trade.side === 'buy') {
115       return 'Long';
116     } else if (trade.side === 'sell') {
117       return 'Short';
118     }
119     // Fallback basado en PnL si no hay side
120     return (trade.pnl ?? 0) >= 0 ? 'Long' : 'Short';
121   }
122
123   getStrategyNameForTrade(trade: GroupedTradeFinal): string {
124     // Si no se siguió estrategia, no mostrar nombre
125     if (!this.selectedDay?.followedStrategy) {
126       return '';
127     }
128
129     if (!this.strategies || this.strategies.length === 0) {
130       return '-';
131     }
132
133     const tradeDate = new Date(Number(trade.lastModified));
134     const today = new Date();
135     today.setHours(23, 59, 59, 999); // 11:59 PM del día actual
136
137     // Buscar la estrategia que estaba activa en la fecha del trade
138     for (const strategy of this.strategies) {
139       // IMPORTANTE: NO filtrar estrategias eliminadas aquí
140       // Las estrategias eliminadas (soft delete) SÍ deben considerarse
141       // porque en el momento del trade existían y podrían haber sido seguidas
142
143       if (strategy.dateActive && strategy.dateActive.length > 0) {
144         // Revisar cada período de activación de esta estrategia
145         for (let i = 0; i < strategy.dateActive.length; i++) {
146           const activeDate = new Date(strategy.dateActive[i]);
147           let inactiveDate: Date;
```

```
148
149           // Si hay fecha de desactivación correspondiente, usarla
150           if (strategy.dateInactive && strategy.dateInactive.length > i) {
151             inactiveDate = new Date(strategy.dateInactive[i]);
152           } else {
153             // No hay fecha de desactivación, verificar si está activa actualmente
154             // Si dateActive tiene más elementos que dateInactive, está activa
155             const isCurrentlyActive = strategy.dateActive.length >
(strategy.dateInactive?.length || 0);
156             if (isCurrentlyActive) {
157               inactiveDate = today;
158             } else {
159               continue; // Esta activación ya fue desactivada
160             }
161           }
162
163           // Verificar si el trade está dentro de este rango de actividad
164           if (tradeDate >= activeDate && tradeDate <= inactiveDate) {
165             return strategy.name;
166           }
167         }
168       }
169     }
170
171     return '-';
172   }
173
174   getTickerColor(ticker: string): string {
175     // Asignar colores a diferentes tickers con transparencia
176     const colors: { [key: string]: string } = {
177       'YM': 'rgba(139, 92, 246, 0.8)',
178       'MYM': 'rgba(139, 92, 246, 0.6)',
179       'ES': 'rgba(59, 130, 246, 0.8)',
180       'NQ': 'rgba(16, 185, 129, 0.8)',
181       'RTY': 'rgba(245, 158, 11, 0.8)'
182     };
183     return colors[ticker] || 'rgba(107, 114, 128, 0.8)';
184   }
185
186   getPnlColor(pnl: number): string {
187     return pnl >= 0 ? '#10B981' : '#EF4444';
188   }
189
190   getStrategyIcon(followed: boolean): string {
191     return followed ? 'icon-check-box' : 'icon-uncheck-box';
192   }
193
194   getStrategyColor(followed: boolean): string {
195     return followed ? '#10B981' : '#EF4444';
196   }
197
198   onClose() {
199     this.close.emit();
200   }
201
202   onSortByTime() {
203     // Implementar lógica de ordenamiento
204     this.trades.sort((a, b) => b.openTime.localeCompare(a.openTime));
205   }
206
207   onFilter() {
208     // Implementar lógica de filtrado
209     console.log('Filter clicked');
210   }
211
212   formatCurrency(value: number): string {
213     return this.numberFormatter.formatCurrency(value);
214   }
215
216   formatPercentage(value: number): string {
217     return this.numberFormatter.formatPercentage(value);
```

```
  }
}

```

Ø=ÜÁ features\report\components\winLossChart

Ø=ÜÄ features\report\components\winLossChart\win-loss-chart.component.ts

```typescript
import { CommonModule } from '@angular/common';
import { Component, Input, OnInit, OnChanges, SimpleChanges, OnDestroy } from '@angular/core';
import { NgApexchartsModule } from 'ng-apexcharts';
import { GroupedTradeFinal } from '../../models/report.model';
import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';

/**
 * Component for displaying win/loss ratio as a donut chart.
 *
 * This component displays a donut chart showing the percentage and monetary value
 * of winning vs losing trades. The chart is responsive and adjusts donut size
 * based on screen width.
 *
 * Features:
 * - Donut chart visualization using ApexCharts
 * - Win/loss percentage calculation
 * - Win/loss monetary value calculation
 * - Responsive design with adaptive donut size
 * - Custom tooltips showing monetary values and percentages
 * - Handles empty data state
 *
 * Relations:
 * - NgApexchartsModule: Chart rendering
 * - NumberFormatterService: Value formatting
 *
 * @component
 * @selector app-win-loss-chart
 * @standalone true
 */
@Component({
  selector: 'app-win-loss-chart',
  templateUrl: './win-loss-chart.component.html',
  styleUrls: ['./win-loss-chart.component.scss'],
  standalone: true,
  imports: [CommonModule, NgApexchartsModule],
})
export class WinLossChartComponent implements OnInit, OnChanges, OnDestroy {
  @Input() values!: GroupedTradeFinal[];

  public chartOptions: any;
  private numberFormatter = new NumberFormatterService();
  public winLossData: {
    winValue: number;
    lossValue: number;
    winPercentage: number;
    lossPercentage: number;
  } = {
    winValue: 0,
    lossValue: 0,
    winPercentage: 0,
    lossPercentage: 0,
  };

  private resizeTimeout?: any;

```

```
  private getDonutSize(): string {
    if (typeof window !== 'undefined') {
      if (window.innerWidth <= 480) {
        return '85%'; // Más grueso en pantallas muy pequeñas para mejor visibilidad
      } else if (window.innerWidth <= 768) {
        return '80%'; // Moderadamente grueso en tablets
      } else if (window.innerWidth <= 1024) {
        return '75%'; // Tamaño medio en pantallas medianas
      }
    }
    return '80%'; // Tamaño normal en desktop
  }

  private getChartSize(): number {
    // El tamaño ahora se controla completamente por CSS
    // Retornamos un valor que será ignorado por ApexCharts
    return 100;
  }

  ngOnInit() {
    this.winLossData = this.calculateWinLossData();
    this.chartOptions = this.getChartOptions();

    // Listener para redimensionar el gráfico con debounce
    if (typeof window !== 'undefined') {
      window.addEventListener('resize', () => {
        if (this.resizeTimeout) {
          clearTimeout(this.resizeTimeout);
        }
        this.resizeTimeout = setTimeout(() => {
          this.chartOptions = this.getChartOptions();
        }, 150);
      });
    }
  }

  ngOnChanges(changes: SimpleChanges) {
    if (changes['values']) {
      // Forzar recálculo y actualización del gráfico
      this.winLossData = this.calculateWinLossData();
      this.chartOptions = this.getChartOptions();

      // Forzar re-renderizado del gráfico después de un pequeño delay
      // Esto es especialmente importante cuando se cargan datos desde localStorage
      setTimeout(() => {
        this.chartOptions = this.getChartOptions();
      }, 100);

      // Segundo intento de actualización para asegurar que se renderice correctamente
      setTimeout(() => {
        this.chartOptions = this.getChartOptions();
      }, 300);
    }
  }

  ngOnDestroy() {
    if (this.resizeTimeout) {
      clearTimeout(this.resizeTimeout);
    }
  }

  getChartOptions(): any {
    // Si no hay datos, mostrar círculo gris sin texto interno
    if (!this.values || this.values.length === 0) {
      return {
        chart: {
          type: 'donut',
          height: '100%',
          width: '100%',
          toolbar: { show: false },
```

```
          foreColor: '#fff',
          fontFamily: 'Inter, Arial, sans-serif',
          background: 'transparent',
        },
        series: [1], // Un solo valor para crear el círculo
        labels: ['No Data'],
        colors: ['#6B7280'], // Color gris
        dataLabels: {
          enabled: false
        },
        plotOptions: {
          pie: {
            donut: {
              size: this.getDonutSize(),
              labels: {
                show: false // No mostrar texto dentro del círculo
              }
            }
          }
        },
        legend: {
          show: false
        },
        tooltip: {
          enabled: false
        },
        stroke: {
          show: false
        }
      };
    }

    return {
      chart: {
        type: 'donut',
        height: '100%',
        width: '100%',
        toolbar: { show: false },
        foreColor: '#fff',
        fontFamily: 'Inter, Arial, sans-serif',
        background: 'transparent',
      },
      series: [this.winLossData.winPercentage, this.winLossData.lossPercentage],
      labels: ['Win', 'Loss'],
      colors: ['#9BF526', '#EC221F'],
      dataLabels: {
        enabled: false
      },
      plotOptions: {
        pie: {
          donut: {
            size: this.getDonutSize(),
            labels: {
              show: false
            }
          }
        }
      },
      legend: {
        show: false
      },
      tooltip: {
        enabled: true,
        custom: ({ series, seriesIndex, dataPointIndex, w }: any) => {
          const percentage = w.globals.seriesPercent[seriesIndex];
          const isWin = seriesIndex === 0; // Ahora el índice 0 es Win
          const color = isWin ? '#468506' : '#EC221F';

          // Usar los valores monetarios reales en lugar del porcentaje
          const moneyValue = isWin ? this.winLossData.winValue : this.winLossData.lossValue;
```

```
          const formattedValue = this.numberFormatter.formatCurrency(moneyValue);
          const formattedPercentage = this.numberFormatter.formatPercentageValue(percentage);

          return `
            <div class="custom-tooltip" style="
              background: rgba(255, 255, 255, 0.95);
              border: 1px solid #e0e0e0;
              border-radius: 8px;
              padding: 12px;
              box-shadow: 0 4px 12px rgba(0, 0, 0, 0.2);
              color: #333;
              font-family: Inter, Arial, sans-serif;
              min-width: 80px;
              text-align: center;
              font-weight: 600;
            ">
              <div style="
                font-size: 16px;
                font-weight: 700;
                color: #333;
                margin-bottom: 4px;
              ">${formattedValue}</div>
              <div style="
                font-size: 12px;
                font-weight: 500;
                color: ${color};
              ">${formattedPercentage}%</div>
            </div>
          `;
        }
      },
      stroke: {
        show: false
      }
    };
  }

  calculateWinLossData() {
    if (!this.values || this.values.length === 0) {
      return { winValue: 0, lossValue: 0, winPercentage: 0, lossPercentage: 0 };
    }

    let totalWinTrades = 0;
    let totalLossTrades = 0;
    let winAmount = 0;
    let lossAmount = 0;

    this.values.forEach(trade => {
      const pnl = trade.pnl ?? 0;
      if (pnl > 0) {
        totalWinTrades++;
        winAmount += pnl;
      } else if (pnl < 0) {
        totalLossTrades++;
        lossAmount += Math.abs(pnl);
      }
    });

    // Calcular solo winValue (plata real ganada) y winPercentage
    const winValue = winAmount; // Plata real ganada
    const totalTrades = totalWinTrades + totalLossTrades;
    const winPercentage = totalTrades > 0 ? (totalWinTrades / totalTrades) * 100 : 0;

    // Calcular lossValue y lossPercentage como lo que falta para completar
    const lossValue = lossAmount; // Plata real perdida
    const lossPercentage = 100 - winPercentage; // Lo que falta para llegar al 100%

    const result = {
      winValue: Math.round(winValue * 100) / 100,
      lossValue: Math.round(lossValue * 100) / 100,
```

```
      winPercentage: Math.round(winPercentage * 10) / 10,
      lossPercentage: Math.round(lossPercentage * 10) / 10
    };

    return result;
  }

  getWinLossData() {
    return this.calculateWinLossData();
  }

  formatCurrency(value: number): string {
    return this.numberFormatter.formatCurrency(value);
  }

  formatPercentage(value: number): string {
    return this.numberFormatter.formatPercentage(value);
  }
}

```

Ø=ÜÁ features\report\models

Ø=ÜÄ features\report\models\report.model.ts

```

import { RuleType } from '../../strategy/models/strategy.model';

/**
 * Interface representing a historical trade record from the trading API.
 *
 * This interface maps the raw array data structure returned by the TradeLocker API
 * into a structured object with named properties for easier access and manipulation.
 *
 * @interface historyTrade
 */
export interface historyTrade {
  id: string;
  tradableInstrumentId: string;
  routeId: string;
  qty: string;
  side: string;
  type: string;
  status: string;
  filledQty: string;
  avgPrice: string;
  price: string;
  stopPrice: string;
  validity: string;
  expireDate: string;
  createdDate: string;
  lastModified: string;
  isOpen: string;
  positionId: string;
  stopLoss: string;
  stopLossType: string;
  takeProfit: string;
  takeProfitType: string;
  strategyId: string;
}

/**
 * Interface representing a grouped trade with position information.
```

```
 *
 * This interface is used for intermediate processing of trades before final grouping.
 * It contains position-level data including entry/exit prices, PnL, and trade status.
 *
 * @interface GroupedTrade
 */
export interface GroupedTrade {
  position_Id: string;
  quantity?: number;
  pnl?: number;
  buy_price?: string;
  sell_price?: string;
  totalSpend?: number;
  updatedAt: string;
  // Nuevas propiedades para claridad
  entryPrice?: string;
  exitPrice?: string;
  side?: string; // 'buy' o 'sell' de la posición
  isWon?: boolean;
  isOpen?: boolean;
  stopLoss?: string;
  takeProfit?: string;
  allTrades?: historyTrade[]; // Todos los trades de esta posición
}

/**
 * Interface representing a final processed trade after grouping by position.
 *
 * This is the final structure used throughout the application to display trade information.
 * It includes all original trade data plus calculated fields like PnL and win status.
 *
 * Used in:
 * - ReportComponent: Main component displaying trades
 * - CalendarComponent: Calendar view of trades
 * - PnlGraphComponent: PnL chart visualization
 * - WinLossChartComponent: Win/loss ratio visualization
 *
 * @interface GroupedTradeFinal
 */
export interface GroupedTradeFinal {
  id: string; // id
  tradableInstrumentId: string; // tradableInstrumentId
  routeId: string; // routeId
  qty: string; // qty
  side: string; // side
  type: string; // type
  status: string; // status
  filledQty: string; // filledQty
  avgPrice: string; // avgPrice
  price: string; // price
  stopPrice: string; // stopPrice
  validity: string; // validity
  expireDate: string; // expireDate
  createdDate: string; // createdDate
  lastModified: string; // lastModified
  isOpen: boolean; // isOpen
  positionId: string; // positionId
  stopLoss: string; // stopLoss
  stopLossType: string; // stopLossType
  takeProfit: string; // takeProfit
  takeProfitType: string; // takeProfitType
  strategyId: string; // strategyId
  instrument?: string; // instrument
  pnl?: number; // pnl
  isWon?: boolean; // isWon
}

/**
 * Interface representing account balance and margin data from the trading API.
 *
```

```
 * Contains comprehensive balance information including available funds, margin requirements,
 * daily trading statistics, and open position data.
 *
 * Used in:
 * - ReportComponent: Displaying account balance information
 * - ReportService: Processing balance data from API
 *
 * @interface BalanceData
 */
export interface BalanceData {
  balance: number, // balance
  projectedBalance: number, // projectedBalance
  availableFunds: number, // availableFunds
  blockedBalance: number, // blockedBalance
  cashBalance: number, // cashBalance
  unsettledCash: number, // unsettledCash
  withdrawalAvailable: number, // withdrawalAvailable
  stocksValue: number, // stocksValue
  optionValue: number, // optionValue
  initialMarginReq: number, // initialMarginReq
  maintMarginReq: number, // maintMarginReq
  marginWarningLevel: number, // marginWarningLevel
  blockedForStocks: number, // blockedForStocks
  stockOrdersReq: number, // stockOrdersReq
  stopOutLevel: number, // stopOutLevel
  warningMarginReq: number, // warningMarginReq
  marginBeforeWarning: number, // marginBeforeWarning
  todayGross: number, // todayGross - A gross profit for today
  todayNet: number, // todayNet - A total profit or loss realized from positions today
  todayFees: number, // todayFees - Fees paid today
  todayVolume: number, // todayVolume - A total volume traded for today
  todayTradesCount: number, // todayTradesCount - A number of trades done for today
  openGrossPnL: number, // openGrossPnL - A profit or loss on all currently opened positions
  openNetPnL: number, // openNetPnL - A net profit or loss on open positions
  positionsCount: number, // positionsCount - A number of currently opened positions
  ordersCount: number // ordersCount - A number of currently placed pending orders
}

/**
 * Interface representing detailed information about a trading instrument.
 *
 * Contains instrument metadata including currency, lot size, trading hours,
 * leverage, and market information.
 *
 * Used in:
 * - ReportService: Fetching instrument details for trade processing
 * - CalendarComponent: Displaying instrument names in calendar view
 *
 * @interface InstrumentDetails
 */
export interface InstrumentDetails {
  barSource: string; // "BID"
  baseCurrency: string; // "XMR"
  betSize: number | null; // null
  betStep: number | null; // null
  bettingCurrency: string | null; // null
  contractMonth: string | null; // null
  country: string | null; // null
  deliveryStatus: string | null; // null
  description: string; // ""
  exerciseStyle: string | null; // null
  firstTradeDate: string | null; // null
  hasDaily: boolean; // true
  hasIntraday: boolean; // true
  industry: string | null; // null
  isin: string; // ""
  lastTradeDate: string | null; // null
  leverage: string; // "2.00"
  localizedName: string; // "XMRUSD"
  logoUrl: string | null; // null
```

```
  lotSize: number; // 10
  lotStep: number; // 0.01
  margin_hedging_type: string; // "none"
  marketCap: number | null; // null
  marketDataExchange: string; // "Cryptos"
  maxLot: number | null; // null
  minLot: number; // 0.01
  name: string; // "XMRUSD"
  noticeDate: string | null; // null
  quotingCurrency: string; // "USD"
  sector: string | null; // null
  settlementDate: string | null; // null
  settlementSystem: string; // "Immediate"
  strikePrice: string | null; // null
  strikeType: string | null; // null
  symbolStatus: string; // "FULLY_OPEN"
  tickCost: Array<{
    leftRangeLimit: number | null;
    tickCost: number;
  }>; // Array(1) [{leftRangeLimit: null, tickCost: 0}]
  tickSize: Array<{
    leftRangeLimit: number | null;
    tickSize: number;
  }>; // Array(1) [{leftRangeLimit: null, tickSize: 0.01}]
  tradeSessionId: number; // 1547
  tradeSessionStatusId: number; // 20
  tradingExchange: string; // "Crypto"
  type: string; // "CRYPTO"
}

/**
 * Interface representing a trading instrument with basic information.
 *
 * Contains essential instrument data including ID, name, routes, and market information.
 * This is a simplified version compared to InstrumentDetails.
 *
 * Used in:
 * - ReportService: Fetching available instruments
 *
 * @interface Instrument
 */
export interface Instrument {
  barSource: string;
  continuous: boolean;
  contractMonth: string;
  country: number;
  description: string;
  hasDaily: boolean;
  hasIntraday: boolean;
  id: number;
  localizedName: string;
  routes: Array<{
    id: string;
    type: string;
  }>;
  logoUrl: string;
  marketDataExchange: string;
  name: string;
  strikePrice: number;
  strikeType: string;
  tradableInstrumentId: number;
  tradingExchange: string;
  type: string;
  underlierId: number;
}

/**
 * Interface representing trading statistics configuration.
 *
 * Contains calculated trading metrics including PnL, win rate, profit factor,
```

```
 * and trade counts. Used to display statistical cards in the report view.
 *
 * Used in:
 * - ReportComponent: Displaying trading statistics
 * - statCardComponent: Individual statistic card display
 *
 * @interface StatConfig
 */
export interface StatConfig {
  netPnl: number;
  tradeWinPercent: number;
  profitFactor: number;
  avgWinLossTrades: number;
  totalTrades: number;
  activePositions: number;
}

/**
 * Interface representing display configuration for trading rules.
 *
 * Contains information about which trading rules are active and should be displayed
 * in the report interface.
 *
 * Used in:
 * - ReportComponent: Managing rule display configuration
 *
 * @interface displayConfigData
 */
export interface displayConfigData {
  title: string;
  type: RuleType;
  isActive: boolean;
}

/**
 * Interface representing a day in the trading calendar.
 *
 * Contains aggregated trade data for a specific day including total PnL,
 * trade count, win percentage, and strategy compliance information.
 *
 * Used in:
 * - CalendarComponent: Calendar view of trades
 * - TradesPopupComponent: Displaying trades for a selected day
 *
 * @interface CalendarDay
 */
export interface CalendarDay {
  date: Date;
  trades: GroupedTradeFinal[];
  pnlTotal: number;
  tradesCount: number;
  followedStrategy: boolean;
  tradeWinPercent: number;
  strategyName?: string | null; // Nombre de la estrategia seguida en este día
  isCurrentMonth: boolean; // Indica si el día pertenece al mes actual
}

/**
 * Interface representing the NgRx store state for the report module.
 *
 * Contains all report-related state including grouped trades, statistics,
 * and user key for API authentication.
 *
 * Used in:
 * - report.reducer.ts: Reducer managing report state
 * - report.selectors.ts: Selectors for accessing report state
 * - report.actions.ts: Actions for updating report state
 *
 * @interface ReportState
 */
```

```
export interface ReportState {
  groupedTrades: GroupedTradeFinal[];
  netPnL: number;
  tradeWin: number;
  profitFactor: number;
  AvgWnL: number;
  totalTrades: number;
  userKey: string;
}

/**
 * Interface representing a monthly trading report.
 *
 * Contains aggregated monthly trading statistics including profit, trades count,
 * and strategy compliance percentage. Used for storing and displaying monthly summaries.
 *
 * Used in:
 * - ReportService: Updating monthly reports
 * - MonthlyReportsService: Managing monthly report data
 *
 * @interface MonthlyReport
 */
export interface MonthlyReport {
  best_trade: string;
  netPnl: number;
  number_trades: number;
  profit: number;
  strategy_followed: number;
  total_spend: number;
  month: number;
  year: number;
  id: string;
}

/**
 * Interface representing plugin usage history record.
 *
 * Tracks when the trading plugin was active or inactive, including date ranges
 * and token requirements. Used to determine strategy compliance for trades.
 *
 * Used in:
 * - CalendarComponent: Determining if trades followed strategies
 * - PluginHistoryService: Managing plugin history data
 *
 * @interface PluginHistoryRecord
 */
export interface PluginHistoryRecord {
  isActive: boolean;
  updatedOn: string;
  id: string;
  tokenNeeded?: boolean;
  dateActive: string[];
  dateInactive: string[];
}

```

Ø=ÜÁ features\report\service

Ø=ÜÄ features\report\service\report.service.ts

```
import { TradeLockerApiService } from '../../../shared/services/tradelocker-api.service';
import { MonthlyReportsService } from '../../../shared/services/monthly-reports.service';
import { Injectable } from '@angular/core';
```

```typescript
import { map, Observable, switchMap } from 'rxjs';
import { AppContextService } from '../../../shared/context';
import {
  GroupedTrade,
  GroupedTradeFinal,
  BalanceData,
  historyTrade,
  MonthlyReport,
  InstrumentDetails,
  Instrument,
} from '../models/report.model';
import {
  arrayToHistoryTrade,
  groupOrdersByPosition,
} from '../utils/normalization-utils';

/**
 * Service for managing report data and API interactions.
 *
 * This service acts as an intermediary between the ReportComponent and external services,
 * handling data fetching, transformation, and context updates for trading reports.
 *
 * Responsibilities:
 * - Fetching trading history from TradeLocker API
 * - Fetching account balance data
 * - Fetching instrument details
 * - Updating monthly reports
 * - Managing loading states in AppContextService
 *
 * Relations:
 * - TradeLockerApiService: Direct API communication
 * - MonthlyReportsService: Monthly report data management
 * - AppContextService: Global state and loading management
 *
 * @injectable
 * @providedIn root
 */
@Injectable({ providedIn: 'root' })
export class ReportService {
  /**
   * Constructor for ReportService.
   *
   * @param tradeLockerApiService - Service for TradeLocker API interactions
   * @param monthlyReportsService - Service for monthly report management
   * @param appContext - Application context service for global state
   */
  constructor(
    private tradeLockerApiService: TradeLockerApiService,
    private monthlyReportsService: MonthlyReportsService,
    private appContext: AppContextService
  ) {}

  /**
   * Updates a monthly report in the database.
   *
   * @param monthlyReport - The monthly report data to update
   * @returns Promise that resolves when the update is complete
   * @memberof ReportService
   */
  async updateMonthlyReport(monthlyReport: MonthlyReport) {
    return this.monthlyReportsService.updateMonthlyReport(monthlyReport);
  }

  /**
   * Gets user authentication key from TradeLocker API.
   *
   * Authenticates user credentials and returns an access token for API requests.
   *
   * @param email - Trading account email
   * @param password - Trading account password
```

```
    * @param server - Trading server name
    * @returns Observable that emits the user key (access token)
    * @memberof ReportService
    */
  getUserKey(
    email: string,
    password: string,
    server: string
  ): Observable<string> {
    return this.tradeLockerApiService.getUserKey(email, password, server);
  }

  /**
   * Fetches trading history data for an account.
   *
   * Retrieves order history from TradeLocker API, transforms it into GroupedTradeFinal objects,
   * and updates the application context with the processed data.
   *
   * Process:
   * 1. Sets loading state in AppContextService
   * 2. Fetches order history from API
   * 3. Transforms array data to historyTrade objects
   * 4. Groups orders by position using groupOrdersByPosition
   * 5. Updates AppContextService with grouped trades
   * 6. Clears loading state
   *
   * Related to:
   * - arrayToHistoryTrade(): Transforms API array to historyTrade
   * - groupOrdersByPosition(): Groups trades by position
   * - AppContextService.updateReportHistory(): Updates global state
   *
   * @param accountId - Trading account ID
   * @param accessToken - User authentication token
   * @param accNum - Account number
   * @returns Observable that emits an array of GroupedTradeFinal objects
   * @memberof ReportService
   */
  getHistoryData(
    accountId: string,
    accessToken: string,
    accNum: number
  ): Observable<GroupedTradeFinal[]> {
    this.appContext.setLoading('report', true);
    this.appContext.setError('report', null);

    return this.tradeLockerApiService.getTradingHistory(accessToken, accountId, accNum)
      .pipe(
        switchMap(async (details) => {
          // Verificar si hay datos válidos
          if (!details || !details.d || !details.d.ordersHistory) {
            this.appContext.updateReportHistory([]);
            this.appContext.setLoading('report', false);
            return [];
          }

          const historyTrades: historyTrade[] =
            details.d.ordersHistory.map(arrayToHistoryTrade);

          // Pasar accessToken y accNum a la función
          const groupedTrades = await groupOrdersByPosition(historyTrades, this, accessToken, accNum);
          // Actualizar contexto con los datos del historial
          this.appContext.updateReportHistory(groupedTrades);
          this.appContext.setLoading('report', false);

          return groupedTrades;
        })
      );
  }

```

```
  /**
   * Fetches account balance data from TradeLocker API.
   *
   * Retrieves comprehensive balance information including available funds, margin
requirements
   * daily trading statistics, and open position data.
   *
   * Process:
   * 1. Sets loading state in AppContextService
   * 2. Fetches balance data from API
   * 3. Maps array data to BalanceData interface
   * 4. Updates AppContextService with balance data
   * 5. Clears loading state
   *
   * Related to:
   * - AppContextService.updateReportBalance(): Updates global balance state
   *
   * @param accountId - Trading account ID
   * @param accessToken - User authentication token
   * @param accNum - Account number
   * @returns Observable that emits BalanceData object
   * @memberof ReportService
   */
  getBalanceData(
    accountId: string,
    accessToken: string,
    accNum: number
  ): Observable<any> {
    this.appContext.setLoading('report', true);
    this.appContext.setError('report', null);

    return this.tradeLockerApiService.getAccountBalance(accountId, accessToken, accNum)
      .pipe(
        map((details) => {
          // Verificar si hay datos válidos
          if (!details || !details.d || !details.d.accountDetailsData) {
            const emptyBalanceData: BalanceData = {
              balance: 0,
              projectedBalance: 0,
              availableFunds: 0,
              blockedBalance: 0,
              cashBalance: 0,
              unsettledCash: 0,
              withdrawalAvailable: 0,
              stocksValue: 0,
              optionValue: 0,
              initialMarginReq: 0,
              maintMarginReq: 0,
              marginWarningLevel: 0,
              blockedForStocks: 0,
              stockOrdersReq: 0,
              stopOutLevel: 0,
              warningMarginReq: 0,
              marginBeforeWarning: 0,
              todayGross: 0,
              todayNet: 0,
              todayFees: 0,
              todayVolume: 0,
              todayTradesCount: 0,
              openGrossPnL: 0,
              openNetPnL: 0,
              positionsCount: 0,
              ordersCount: 0
            };

            this.appContext.updateReportBalance(emptyBalanceData);
            this.appContext.setLoading('report', false);
            return emptyBalanceData;
          }

          // Extract all balance data for calculations
```

```
          const accountData = details.d;

          // Mapear el array accountDetailsData a las propiedades específicas
          const accountDetailsData = accountData.accountDetailsData;
          const balanceData: BalanceData = {
            balance: accountDetailsData[0] || 0,
            projectedBalance: accountDetailsData[1] || 0,
            availableFunds: accountDetailsData[2] || 0,
            blockedBalance: accountDetailsData[3] || 0,
            cashBalance: accountDetailsData[4] || 0,
            unsettledCash: accountDetailsData[5] || 0,
            withdrawalAvailable: accountDetailsData[6] || 0,
            stocksValue: accountDetailsData[7] || 0,
            optionValue: accountDetailsData[8] || 0,
            initialMarginReq: accountDetailsData[9] || 0,
            maintMarginReq: accountDetailsData[10] || 0,
            marginWarningLevel: accountDetailsData[11] || 0,
            blockedForStocks: accountDetailsData[12] || 0,
            stockOrdersReq: accountDetailsData[13] || 0,
            stopOutLevel: accountDetailsData[14] || 0,
            warningMarginReq: accountDetailsData[15] || 0,
            marginBeforeWarning: accountDetailsData[16] || 0,
            todayGross: accountDetailsData[17] || 0,
            todayNet: accountDetailsData[18] || 0,
            todayFees: accountDetailsData[19] || 0,
            todayVolume: accountDetailsData[20] || 0,
            todayTradesCount: accountDetailsData[21] || 0,
            openGrossPnL: accountDetailsData[22] || 0,
            openNetPnL: accountDetailsData[23] || 0,
            positionsCount: accountDetailsData[24] || 0,
            ordersCount: accountDetailsData[25] || 0
          };

          // Actualizar contexto con los datos de balance
          this.appContext.updateReportBalance(balanceData);
          this.appContext.setLoading('report', false);

          return balanceData;
        })
      );
  }

  /**
   * Fetches detailed information about a trading instrument.
   *
   * Retrieves instrument metadata including lot size, name, currency, and trading specifications.
   * Used primarily for calculating accurate PnL and displaying instrument names.
   *
   * @param accessToken - User authentication token
   * @param tradableInstrumentId - Unique identifier for the instrument
   * @param routeId - Route ID for the instrument
   * @param accNum - Account number
   * @returns Observable that emits InstrumentDetails object
   * @memberof ReportService
   */
  getInstrumentDetails(
    accessToken: string,
    tradableInstrumentId: string,
    routeId: string,
    accNum: number
  ): Observable<InstrumentDetails> {
    return this.tradeLockerApiService.getInstrumentDetails(accessToken, tradableInstrumentId, routeId, accNum)
      .pipe(
        map((details) => {
          // Extract all instrument data for calculations
          const instrumentData = details.d;
          const instrumentDetailsData: InstrumentDetails = instrumentData;

          return instrumentDetailsData;
        })
```

```
      );
  }

  /**
   * Fetches all available trading instruments for an account.
   *
   * Retrieves a list of all instruments that can be traded on the account,
   * including basic information like ID, name, and routes.
   *
   * @param accessToken - User authentication token
   * @param accNum - Account number
   * @param accountId - Trading account ID
   * @returns Observable that emits an array of Instrument objects
   * @memberof ReportService
   */
  getAllInstruments(
    accessToken: string,
    accNum: number,
    accountId: string
  ): Observable<Instrument[]> {
    return this.tradeLockerApiService.getAllInstruments(accessToken, accountId, accNum)
      .pipe(
        map((details) => {
          return details.d.instruments;
        })
      )
      .pipe(
        map((instruments) => {
          return instruments.map((instrument: Instrument) => {
            return instrument;
          });
        })
      );
  }
}
```

Ø=ÜÁ features\report\store

Ø=ÜÄ features\report\store\report.actions.ts

```
import { createAction, props } from '@ngrx/store';
import { GroupedTrade, GroupedTradeFinal } from '../models/report.model';

/**
 * Action to trigger fetching report history trades.
 *
 * @action getReportHistory
 */
export const getReportHistory = createAction(
  '[Report] get report history trades',
  props<{ userId: string }>()
);

/**
 * Action to get user authentication key.
 *
 * @action getUserKey
 */
export const getUserKey = createAction(
  '[Report] Get User Key',
  props<{ email: string; password: string; server: string }>()
);
```

```
23
24  /**
25   * Action to set user authentication key in store.
26   *
27   * @action setUserKey
28   */
29  export const setUserKey = createAction(
30    '[Report] Set User Key',
31    props<{ userKey: string }>()
32  );
33
34  /**
35   * Action to set grouped trades in store.
36   *
37   * @action setGroupedTrades
38   */
39  export const setGroupedTrades = createAction(
40    '[Report] Set Grouped Trades',
41    props<{ groupedTrades: GroupedTradeFinal[] }>()
42  );
43
44  /**
45   * Action to set net PnL value in store.
46   *
47   * @action setNetPnL
48   */
49  export const setNetPnL = createAction(
50    '[Report] Set Net PnL',
51    props<{ netPnL: number }>()
52  );
53
54  /**
55   * Action to set trade win percentage in store.
56   *
57   * @action setTradeWin
58   */
59  export const setTradeWin = createAction(
60    '[Report] Set Trade Win',
61    props<{ tradeWin: number }>()
62  );
63
64  /**
65   * Action to set profit factor value in store.
66   *
67   * @action setProfitFactor
68   */
69  export const setProfitFactor = createAction(
70    '[Report] Set Profit Factor',
71    props<{ profitFactor: number }>()
72  );
73
74  /**
75   * Action to set average win/loss ratio in store.
76   *
77   * @action setAvgWnL
78   */
79  export const setAvgWnL = createAction(
80    '[Report] Set Avg Win/Loss',
81    props<{ avgWnL: number }>()
82  );
83
84  /**
85   * Action to set total trades count in store.
86   *
87   * @action setTotalTrades
88   */
89  export const setTotalTrades = createAction(
90    '[Report] Set Total Trades',
91    props<{ totalTrades: number }>()
92  );
```

Ø=ÜÄ features\report\store\report.reducer.ts

```ts
import { createReducer, on } from '@ngrx/store';

import { ReportState } from '../models/report.model';
import {
  getUserKey,
  setAvgWnL,
  setGroupedTrades,
  setNetPnL,
  setProfitFactor,
  setTotalTrades,
  setTradeWin,
  setUserKey,
} from './report.actions';

/**
 * Initial state for the report feature.
 *
 * @constant initialReportState
 */
export const initialReportState: ReportState = {
  groupedTrades: [],
  netPnL: 0,
  tradeWin: 0,
  profitFactor: 0,
  AvgWnL: 0,
  totalTrades: 0,
  userKey: '',
};

/**
 * Reducer for managing report state.
 *
 * Handles all report-related actions and updates the state accordingly.
 * Actions handled:
 * - setUserKey: Updates user authentication key
 * - setGroupedTrades: Updates grouped trades array
 * - setNetPnL: Updates net PnL value
 * - setTradeWin: Updates trade win percentage
 * - setProfitFactor: Updates profit factor
 * - setAvgWnL: Updates average win/loss ratio
 * - setTotalTrades: Updates total trades count
 *
 * @reducer reportReducer
 */
export const reportReducer = createReducer(
  initialReportState,
  on(setUserKey, (state, { userKey }) => ({
    ...state,
    userKey,
  })),
  on(setGroupedTrades, (state, { groupedTrades }) => ({
    ...state,
    groupedTrades,
  })),
  on(setNetPnL, (state, { netPnL }) => ({
    ...state,
    netPnL,
  })),
  on(setTradeWin, (state, { tradeWin }) => ({
    ...state,
    tradeWin,
  })),
```

```
  on(setProfitFactor, (state, { profitFactor }) => ({
    ...state,
    profitFactor,
  })),
  on(setAvgWnL, (state, { avgWnL }) => ({
    ...state,
    AvgWnL: avgWnL,
  })),
  on(setTotalTrades, (state, { totalTrades }) => ({
    ...state,
    totalTrades,
  }))
);

```

Ø=ÜÄ features\report\store\report.selectors.ts

```
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { ReportState } from '../models/report.model';

/**
 * Feature selector for the report state.
 *
 * @selector selectReport
 */
export const selectReport = createFeatureSelector<ReportState>('report');

/**
 * Selector for grouped trades from report state.
 *
 * @selector selectGroupedTrades
 */
export const selectGroupedTrades = createSelector(
  selectReport,
  (state: ReportState) => state.groupedTrades
);

/**
 * Selector for user key from report state.
 *
 * @selector selectUserKey
 */
export const selectUserKey = createSelector(
  selectReport,
  (state: ReportState) => state.userKey
);

/**
 * Selector for net PnL from report state.
 *
 * @selector selectNetPnL
 */
export const selectNetPnL = createSelector(
  selectReport,
  (state: ReportState) => state.netPnL
);

/**
 * Selector for trade win percentage from report state.
 *
 * @selector selectTradeWin
 */
export const selectTradeWin = createSelector(
  selectReport,
  (state: ReportState) => state.tradeWin
);
```

```
50
51  /**
52   * Selector for profit factor from report state.
53   *
54   * @selector selectProfitFactor
55   */
56  export const selectProfitFactor = createSelector(
57    selectReport,
58    (state: ReportState) => state.profitFactor
59  );
60
61  /**
62   * Selector for average win/loss ratio from report state.
63   *
64   * @selector selectAvgWnL
65   */
66  export const selectAvgWnL = createSelector(
67    selectReport,
68    (state: ReportState) => state.AvgWnL
69  );
70
71  /**
72   * Selector for total trades count from report state.
73   *
74   * @selector selectTotalTrades
75   */
76  export const selectTotalTrades = createSelector(
77    selectReport,
78    (state: ReportState) => state.totalTrades
79  );
80
```

Ø=ÜÁ features\report\utils

Ø=ÜÄ features\report\utils\firebase-data-utils.ts

```
1   import { GroupedTrade, GroupedTradeFinal } from '../models/report.model';
2
3   /**
4    * Gets the best trade (highest PnL) from an array of trades.
5    *
6    * Iterates through all trades and finds the one with the maximum PnL value.
7    * Returns null if there are no trades or if all trades have undefined PnL.
8    *
9    * @param groupedTrades - Array of GroupedTradeFinal objects
10   * @returns The highest PnL value rounded to nearest integer, or null if no valid trades
11   */
12  export const getBestTrade = (groupedTrades: GroupedTradeFinal[]): number | null => {
13    if (!groupedTrades || groupedTrades.length === 0) {
14      return null;
15    }
16
17    const tradeWithMaxPnl = groupedTrades.reduce((maxTrade, currentTrade) => {
18      if (currentTrade.pnl === undefined) return maxTrade;
19      if (maxTrade.pnl === undefined || currentTrade.pnl > maxTrade.pnl) {
20        return currentTrade;
21      }
22      return maxTrade;
23    }, groupedTrades[0]);
24
25    return Math.round(tradeWithMaxPnl.pnl ?? 0);
26  };
27
```

```
/**
 * Calculates the total amount spent on trades.
 *
 * Multiplies price by quantity for each trade and sums the results.
 * Used for calculating total investment or capital used in trading.
 *
 * @param groupedTrades - Array of GroupedTradeFinal objects
 * @returns Total spend amount rounded down to nearest integer
 */
export const getTotalSpend = (groupedTrades: GroupedTradeFinal[]): number | null => {
  const totalSpend = groupedTrades.reduce((total, trade) => {
    const price = Number(trade.price) || 0;
    const qty = Number(trade.qty) || 0;
    return total + (price * qty);
  }, 0);
  return Math.floor(totalSpend);
};

/**
 * Generates a unique ID for monthly report data.
 *
 * Creates an ID by combining a base ID with formatted month and year.
 * Format: "{baseId}-{MM}-{YYYY}"
 *
 * @param baseId - Base identifier (typically user ID or account ID)
 * @param month - Month number (1-12)
 * @param year - Year number (e.g., 2024)
 * @returns Formatted ID string
 */
export function newDataId(baseId: string, month: number, year: number): string {
  const monthFormatted = month.toString().padStart(2, '0');
  return `${baseId}-${monthFormatted}-${year}`;
}

```

Ø=ÜÄ features\report\utils\normalization-utils.ts

```
import { GroupedTrade, GroupedTradeFinal, historyTrade } from '../models/report.model';

/**
 * Converts an array from the TradeLocker API into a historyTrade object.
 *
 * The API returns trade data as arrays where each index corresponds to a specific field.
 * This function maps the array indices to named properties for easier access.
 *
 * Array mapping:
 * - [0]: id
 * - [1]: tradableInstrumentId
 * - [2]: routeId
 * - [3]: qty
 * - [4]: side
 * - [5]: type
 * - [6]: status
 * - [7]: filledQty
 * - [8]: avgPrice
 * - [9]: price
 * - [10]: stopPrice
 * - [11]: validity
 * - [12]: expireDate
 * - [13]: createdDate
 * - [14]: lastModified
 * - [15]: isOpen
 * - [16]: positionId
 * - [17]: stopLoss
 * - [18]: stopLossType
 * - [19]: takeProfit
```

```
 * - [20]: takeProfitType
 * - [21]: strategyId
 *
 * @param arr - Array from TradeLocker API containing trade data
 * @returns historyTrade object with named properties
 */
export function arrayToHistoryTrade(arr: any[]): historyTrade {
  // Mapeo según la configuración de la API:
  // 0: id, 1: tradableInstrumentId, 2: routeId, 3: qty, 4: side, 5: type, 6: status, 7: filledQty, 8: avgPrice, 9: price, 10: stopPrice, 11: validity, 12: expireDate, 13: createdDate, 14: lastModified, 15: isOpen, 16: positionId, 17: stopLoss, 18: stopLossType, 19: takeProfit, 20: takeProfitType, 21: strategyId
  return {
    id: arr[0], // id
    tradableInstrumentId: arr[1], // tradableInstrumentId
    routeId: arr[2], // routeId
    qty: arr[3], // qty
    side: arr[4], // side
    type: arr[5], // type
    status: arr[6], // status
    filledQty: arr[7], // filledQty
    avgPrice: arr[8], // avgPrice
    price: arr[9], // price
    stopPrice: arr[10], // stopPrice
    validity: arr[11], // validity
    expireDate: arr[12], // expireDate
    createdDate: arr[13], // createdDate
    lastModified: arr[14], // lastModified
    isOpen: arr[15], // isOpen
    positionId: arr[16], // positionId
    stopLoss: arr[17], // stopLoss
    stopLossType: arr[18], // stopLossType
    takeProfit: arr[19], // takeProfit
    takeProfitType: arr[20], // takeProfitType
    strategyId: arr[21], // strategyId
  };
}

/**
 * Groups trading orders by position ID and calculates PnL for each position.
 *
 * This function processes raw order history and groups orders that belong to the same position
 * (identified by positionId). It then calculates the PnL for each position by matching
 * opening and closing orders, and fetches instrument details to ensure accurate calculations.
 *
 * Process:
 * 1. Filters valid orders (status 'Filled' and valid positionId)
 * 2. Groups orders by positionId
 * 3. Fetches instrument details for all unique instruments
 * 4. For each position:
 *    - Finds opening order (isOpen: 'true')
 *    - Finds closing order (isOpen: 'false')
 *    - Calculates PnL based on entry/exit prices and lot size
 *    - Determines if trade was won or lost
 * 5. Returns array of GroupedTradeFinal objects
 *
 * Related to:
 * - fetchInstrumentDetails(): Fetches lot size and instrument names
 * - ReportService.getInstrumentDetails(): API call for instrument data
 *
 * @param orders - Array of historyTrade objects to group
 * @param reportService - ReportService instance for fetching instrument details
 * @param accessToken - User authentication token
 * @param accNum - Account number
 * @returns Promise that resolves to an array of GroupedTradeFinal objects
 */
export async function groupOrdersByPosition(orders: historyTrade[], reportService: any, accessToken: string, accNum: number): Promise<GroupedTradeFinal[]> {
  // Filtrar solo trades que no estén cancelados y que tengan status 'Filled' (case insensitive)
  const validOrders = orders.filter(order => {
    const hasValidStatus = order.status && (
      order.status.toLowerCase() === 'filled' ||
      order.status === 'Filled' ||
```

```
      order.status === 'filled'
    );
    const hasValidPositionId = order.positionId &&
      order.positionId !== 'null' &&
      order.positionId !== '' &&
      order.positionId !== null &&
      order.positionId !== undefined;

    return hasValidStatus && hasValidPositionId;
  });

  // Agrupar órdenes por positionId
  const ordersByPosition: { [positionId: string]: historyTrade[] } = {};

  validOrders.forEach(order => {
    const positionId = order.positionId;
    if (!ordersByPosition[positionId]) {
      ordersByPosition[positionId] = [];
    }
    ordersByPosition[positionId].push(order);
  });

  // Obtener lotSize de todos los instrumentos únicos ANTES de procesar trades
  const instrumentDetailsMap = await fetchInstrumentDetails(validOrders, reportService,
accessToken, accNum);
  // Procesar cada posición para crear un trade final
  const finalTrades: GroupedTradeFinal[] = [];
  let totalWins = 0;
  let totalLosses = 0;

  Object.entries(ordersByPosition).forEach(([positionId, positionOrders]) => {
    // Ordenar por fecha de creación para asegurar orden correcto
    positionOrders.sort((a, b) => new Date(a.createdDate).getTime() - new
Date(b.createdDate).getTime());
    // Buscar la orden de apertura (isOpen: true) y la de cierre (isOpen: false)
    const openOrder = positionOrders.find(order => order.isOpen === 'true');
    const closeOrder = positionOrders.find(order => order.isOpen === 'false');

    if (openOrder && closeOrder) {
      // Calcular P&L
      const tradeKey = `${openOrder.tradableInstrumentId}-${openOrder.routeId}`;
      const instrumentDetails = instrumentDetailsMap.get(tradeKey);
      const lotSize = instrumentDetails?.lotSize || 1;

      const entryPrice = Number(openOrder.price);
      const exitPrice = Number(closeOrder.price);
      const quantity = Number(openOrder.qty);

      // Calcular P&L según el tipo de trade
      let pnl: number;
      let isWin = false;

      if (openOrder.side === 'buy' && closeOrder.side === 'sell') {
        // BUY !' SELL: Ganas si vendes más caro de lo que compraste
        pnl = (exitPrice - entryPrice) * (quantity * lotSize);
        isWin = exitPrice > entryPrice;
      } else if (openOrder.side === 'sell' && closeOrder.side === 'buy') {
        // SELL !' BUY: Ganas si compras más barato de lo que vendiste
        pnl = (entryPrice - exitPrice) * (quantity * lotSize);
        isWin = entryPrice > exitPrice;
      } else {
        // Fallback (no debería pasar)
        pnl = (exitPrice - entryPrice) * (quantity * lotSize);
        isWin = pnl > 0;
      }

      // Crear el trade final usando la orden de apertura como base
      const finalTrade: GroupedTradeFinal = {
        ...openOrder,
        instrument: instrumentDetails?.name || openOrder.tradableInstrumentId,
```

```
        pnl: pnl,
        isWon: isWin,
        isOpen: false, // Ya está cerrado
        lastModified: closeOrder.lastModified, // Usar la fecha de cierre
      };

      // Validar que el trade final tenga positionId válido
      if (finalTrade.positionId && finalTrade.positionId !== 'null' &&
finalTrade.positionId !== '') {
        finalTrades.push(finalTrade);
      }

      // Contar wins y losses
      if (isWin) {
        totalWins++;
      } else {
        totalLosses++;
      }
    } else {
      // Si no se encuentran ambos trades, agregar la orden de apertura si existe
      if (openOrder) {
        const tradeKey = `${openOrder.tradableInstrumentId}-${openOrder.routeId}`;
        const instrumentDetails = instrumentDetailsMap.get(tradeKey);

        const finalTrade: GroupedTradeFinal = {
          ...openOrder,
          instrument: instrumentDetails?.name || openOrder.tradableInstrumentId,
          pnl: 0,
          isWon: false,
          isOpen: true,
        };

        // Validar que el trade final tenga positionId válido
        if (finalTrade.positionId && finalTrade.positionId !== 'null' &&
finalTrade.positionId !== '') {
          finalTrades.push(finalTrade);
        }
      }
    }
  });

  return finalTrades;
}

/**
 * Fetches instrument details for all unique instruments in the orders.
 *
 * This helper function extracts unique instrument combinations (tradableInstrumentId +
routeId)
 * from orders and fetches their details (lot size and name) from the API.
 * It processes instruments sequentially with a small delay to avoid rate limiting.
 *
 * @param orders - Array of historyTrade objects
 * @param reportService - ReportService instance for API calls
 * @param accessToken - User authentication token
 * @param accNum - Account number
 * @returns Promise that resolves to a Map with instrument keys and their details
 * @private
 */
async function fetchInstrumentDetails(
  orders: historyTrade[],
  reportService: any,
  accessToken: string,
  accNum: number
): Promise<Map<string, { lotSize: number, name: string }>> {
  // Extraer tradableInstrumentId y routeId únicos
  const uniqueInstruments = new Map<string, { tradableInstrumentId: string, routeId: string }
>();
  orders.forEach(order => {
    if (order.tradableInstrumentId && order.routeId) {
      const key = `${order.tradableInstrumentId}-${order.routeId}`;
      if (!uniqueInstruments.has(key)) {
        uniqueInstruments.set(key, {
```

```
          tradableInstrumentId: order.tradableInstrumentId,
          routeId: order.routeId
        });
      }
    }
  });

  // Hacer consultas a la API para obtener lotSize y name de cada instrumento
  const instrumentDetailsMap = new Map<string, { lotSize: number, name: string }>(); // key:
tradableInstrumentId-routeId, value: {lotSize, name}
  // Procesar cada instrumento único de forma secuencial
  for (const [key, instrument] of uniqueInstruments) {
    try {
      // Hacer petición individual para cada instrumento
      const instrumentDetails = await reportService.getInstrumentDetails(
        accessToken,
        instrument.tradableInstrumentId,
        instrument.routeId,
        accNum
      ).toPromise();

      const lotSize = instrumentDetails.lotSize;
      const name = instrumentDetails.name;

      instrumentDetailsMap.set(key, { lotSize, name });

      // Pequeña pausa entre peticiones para evitar rate limiting
      await new Promise(resolve => setTimeout(resolve, 100));

    } catch (error) {
      console.error(`'L Error querying instrument ${key}:`, error);
      instrumentDetailsMap.set(key, { lotSize: 1, name:
instrument.tradableInstrumentId }); // Default values si falla la consulta
    }
  }

  return instrumentDetailsMap;
}


/**
 * Calculates the total net PnL from an array of trades.
 *
 * Sums all PnL values from trades and rounds to the nearest integer.
 *
 * @param trades - Array of trade objects with optional pnl property
 * @returns Total net PnL rounded to nearest integer
 */
export function calculateNetPnl(trades: { pnl?: number }[]): number {
  const total = trades.reduce((sum, t) => sum + (t.pnl ?? 0), 0);
  return Math.round(total);
}

/**
 * Calculates the percentage of winning trades.
 *
 * Counts trades with positive PnL and calculates the percentage
 * relative to total trades. Returns 0 if there are no trades.
 *
 * @param trades - Array of trade objects with optional pnl property
 * @returns Win percentage rounded to nearest integer (0-100)
 */
export function calculateTradeWinPercent(trades: { pnl?: number }[]): number {
  const wins = trades.filter((t) => t.pnl !== undefined && t.pnl > 0).length;
  return trades.length > 0 ? Math.round((wins / trades.length) * 100) : 0;
}

/**
 * Calculates the profit factor from an array of trades.
 *
 * Profit factor is calculated as: gross profit / gross loss
```

```
 * - If there are no losses, returns 999.99 (infinite profit factor)
 * - If there are no profits, returns 0
 * - Otherwise, returns the ratio rounded to 2 decimal places
 *
 * @param trades - Array of trade objects with optional pnl property
 * @returns Profit factor rounded to 2 decimal places
 */
export function calculateProfitFactor(trades: { pnl?: number }[]): number {
  const grossProfit = trades
    .filter((t) => (t.pnl ?? 0) > 0)
    .reduce((sum, t) => sum + t.pnl!, 0);
  const grossLoss = trades
    .filter((t) => (t.pnl ?? 0) < 0)
    .reduce((sum, t) => sum + Math.abs(t.pnl!), 0);

  // If there are no losses, profit factor should be calculated differently
  if (grossLoss === 0) {
    // If there are profits and no losses, return a high but finite number
    return grossProfit > 0 ? 999.99 : 0;
  }

  const profitFactor = grossProfit / grossLoss;
  return Math.round(profitFactor * 100) / 100;
}

/**
 * Calculates the average win/loss ratio.
 *
 * Calculates the ratio of average winning trade to average losing trade.
 * - If there are no losses, returns 999.99 (infinite ratio)
 * - If there are no wins, returns 0
 * - Otherwise, returns the ratio rounded to 2 decimal places
 *
 * @param trades - Array of trade objects with optional pnl property
 * @returns Average win/loss ratio rounded to 2 decimal places
 */
export function calculateAvgWinLossTrades(trades: { pnl?: number }[]): number {
  const wins = trades.filter((t) => t.pnl !== undefined && t.pnl > 0);
  const losses = trades.filter((t) => t.pnl !== undefined && t.pnl < 0);

  // If there are no losses, we can't calculate a ratio
  if (losses.length === 0) {
    // If there are wins but no losses, return a high ratio
    return wins.length > 0 ? 999.99 : 0;
  }

  const grossProfit = wins.reduce((sum, t) => sum + t.pnl!, 0);
  const grossLoss = losses.reduce((sum, t) => sum + Math.abs(t.pnl!), 0);

  const avgWin = wins.length > 0 ? grossProfit / wins.length : 0;
  const avgLoss = losses.length > 0 ? grossLoss / losses.length : 0;

  return avgLoss > 0 ? Math.round((avgWin / avgLoss) * 100) / 100 : 0;
}

/**
 * Calculates the total number of trades.
 *
 * Simply returns the length of the trades array.
 *
 * @param trades - Array of trade objects
 * @returns Total number of trades
 */
export function calculateTotalTrades(trades: any[]): number {
  return trades.length;
}

/**
 * Groups trades by month and calculates total PnL per month.
 *
```

```
 * Processes trades and aggregates PnL values by month (format: "YYYY-MM").
 * Used for generating monthly PnL charts and reports.
 *
 * @param trades - Array of GroupedTrade objects with updatedAt and pnl properties
 * @returns Object with month keys (YYYY-MM) and total PnL values
 */
export function getMonthlyPnL(trades: GroupedTrade[]): {
  [month: string]: number;
} {
  const monthlyPnL: { [month: string]: number } = {};

  trades.forEach((trade) => {
    const d = new Date(trade.updatedAt);
    const key = `${d.getFullYear()}-${String(d.getMonth() + 1).padStart(
      2,
      '0'
    )}`;
    monthlyPnL[key] = (monthlyPnL[key] ?? 0) + (trade.pnl ?? 0);
  });

  return monthlyPnL;
}
```

Ø=ÜÁ features\revenue

Ø=ÜÄ features\revenue\revenue.component.ts

```
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { LoadingPopupComponent } from '../../shared/pop-ups/loading-pop-up/loading-popup.component';
import { Component } from '@angular/core';
import {
  DailyRevenueData,
  MonthlyRevenueData,
  OrderTableRow,
  RevenueSummary,
  RevenueTableRow,
  SubscriptionTableRow,
} from './models/revenue';
import { Store } from '@ngrx/store';
import {
  dailyRevenueMock,
  mockRevenueSummary,
  monthlyRevenueMock,
  orderTableMock,
  revenueTableMock,
  subscriptionTableMock,
} from './mocks/revenue_mock';
import { statCardComponent } from '../report/components/statCard/stat_card.component';
import { RevenueGraphComponent } from './components/revenueGraph/revenue-graph.component';
import { RevenueTableComponent } from './components/revenue-table/revenue-table.component';
import { OrdersTableComponent } from './components/orders-table/orders-table.component';
import { SubscriptionsTableComponent } from './components/subscriptions-table/subscriptions-table.component';
import { ReportService } from '../report/service/report.service';
import { selectUser } from '../auth/store/user.selectios';
import { User } from '../overview/models/overview';
import { AccountData } from '../auth/models/userModel';

/**
 * Main component for displaying revenue analytics and data.
 *
 * This component displays revenue-related information including:
```

```
 * - Revenue summary statistics (gross revenue, returns, coupons, net revenue)
 * - Revenue charts (daily and monthly)
 * - Revenue table with filtering and pagination
 * - Orders table with filtering and pagination
 * - Subscriptions table with filtering and pagination
 *
 * Currently uses mock data for display. Future implementation will fetch
 * real data from APIs based on user accounts and access tokens.
 *
 * Relations:
 * - RevenueGraphComponent: Displays revenue charts
 * - RevenueTableComponent: Displays revenue table
 * - OrdersTableComponent: Displays orders table
 * - SubscriptionsTableComponent: Displays subscriptions table
 * - ReportService: For fetching user keys and historical data (future implementation)
 * - Store (NgRx): For accessing user data
 *
 * @component
 * @selector app-revenue
 * @standalone true
 */
@Component({
  selector: 'app-revenue',
  imports: [
    CommonModule,
    LoadingPopupComponent,
    FormsModule,
    statCardComponent,
    RevenueGraphComponent,
    RevenueTableComponent,
    OrdersTableComponent,
    SubscriptionsTableComponent,
  ],
  templateUrl: './revenue.component.html',
  styleUrl: './revenue.component.scss',
  standalone: true,
})
export class RevenueComponent {
  revenueSummary: RevenueSummary | null = null;
  revenueDailyData: DailyRevenueData[] | null = null;
  revenueMonthlyData: MonthlyRevenueData[] | null = null;
  revenueTableData: RevenueTableRow[] | null = null;
  loading = false;
  orderTableData: OrderTableRow[] | null = null;
  subscriptionsTableData: SubscriptionTableRow[] | null = null;

  // Propiedades para el accessToken
  accessToken: string | null = null;
  user: User | null = null;
  accountsData: AccountData[] = [];

  constructor(private store: Store, private reportService: ReportService) {}

  /**
   * Initializes the component on load.
   *
   * Loads configuration (mock data) and fetches user data from the store.
   *
   * @memberof RevenueComponent
   */
  ngOnInit(): void {
    this.loadConfig();
    this.getUserData();
  }

  /**
   * Loads configuration data (currently using mock data).
   *
   * Initializes all revenue-related data from mock sources:
   * - Revenue summary
```

```
   * - Daily and monthly revenue data
   * - Revenue table data
   * - Orders table data
   * - Subscriptions table data
   *
   * NOTE: In production, this should fetch data from APIs.
   *
   * @memberof RevenueComponent
   */
  loadConfig() {
    this.revenueSummary = mockRevenueSummary;
    this.revenueDailyData = dailyRevenueMock;
    this.revenueMonthlyData = monthlyRevenueMock;
    this.revenueTableData = revenueTableMock;
    this.orderTableData = orderTableMock;
    this.subscriptionsTableData = subscriptionTableMock;
  }

  /**
   * Fetches user data from the NgRx store.
   *
   * Subscribes to the selectUser selector to get current user information.
   * If user has trading accounts, attempts to fetch access token for API calls.
   * Currently falls back to mock data if no accounts are available.
   *
   * Related to:
   * - Store.select(selectUser): Gets user from NgRx store
   * - fetchUserKey(): Fetches access token for API calls
   *
   * @memberof RevenueComponent
   */
  getUserData() {
    // Obtener datos del usuario desde el store
    this.store.select(selectUser).subscribe((userState) => {
      if (userState && userState.user) {
        this.user = userState.user;
        // Por ahora, usar datos mock ya que el modelo User no tiene accountsData
        // TODO: Agregar accountsData al modelo User o obtenerlo de otra fuente
        this.accountsData = []; // Temporalmente vacío

        // Si hay cuentas, obtener el accessToken de la primera cuenta
        if (this.accountsData.length > 0) {
          this.fetchUserKey(this.accountsData[0]);
        } else {
          console.warn('No hay cuentas de trading disponibles - usando datos mock');
          this.orderTableData = orderTableMock;
        }
      }
    });
  }

  /**
   * Fetches user authentication key for API access.
   *
   * Uses ReportService to authenticate with trading account credentials
   * and obtain an access token. On success, triggers fetching historical data.
   *
   * Related to:
   * - ReportService.getUserKey(): Authenticates and gets access token
   * - getHistoricalData(): Fetches historical data after authentication
   *
   * @param account - Trading account data with credentials
   * @memberof RevenueComponent
   */
  fetchUserKey(account: AccountData) {
    this.reportService
      .getUserKey(
        account.emailTradingAccount,
        account.brokerPassword,
        account.server
```

```
      )
      .subscribe({
        next: (accessToken: string) => {
          this.accessToken = accessToken;
          this.getHistoricalData();
        },
        error: (error) => {
          console.error('Error al obtener el accessToken:', error);
          this.orderTableData = orderTableMock;
        }
      });
  }

  /**
   * Fetches historical revenue data from the API.
   *
   * Currently a placeholder method. In production, this should:
   * - Use the access token to authenticate API requests
   * - Fetch historical order and revenue data
   * - Update orderTableData with real data
   *
   * NOTE: This method is not yet fully implemented.
   *
   * @memberof RevenueComponent
   */
  getHistoricalData() {
    if (!this.accessToken || this.accountsData.length === 0) {
      console.warn('No hay accessToken o cuentas disponibles');
      this.orderTableData = orderTableMock;
      return;
    }

    // Valores de ejemplo para probar el endpoint
    // TODO: Reemplazar con valores reales
    const routeId = 1;
    const from = Date.now() - (30 * 24 * 60 * 60 * 1000); // 30 días atrás
    const to = Date.now(); // Ahora
    const resolution = '1D'; // Diario
    const tradableInstrumentId = 1; // TODO: Usar ID real del instrumento
    const accNum = this.accountsData[0].accountNumber || 1; // Usar número de cuenta real


  }
}

```

Ø=ÜÁ features\revenue\components\orders-table

Ø=ÜÄ features\revenue\components\orders-table\orders-table.component.ts

```
import { Component, Input, Output } from '@angular/core';
import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
import {
  OrderFilter,
  OrderStatus,
  OrderTableRow,
  RevenueFilter,
  RevenueTableRow,
} from '../../models/revenue';

/**
```

```
 * Component for displaying orders data in a table format.
 *
 * This component provides a paginated and filterable table for order data.
 * It supports filtering by search term, order status, and total amount,
 * as well as sorting and pagination.
 *
 * Features:
 * - Search by order ID or user name
 * - Filter by order status (Completed, Pending, Cancelled, Failed)
 * - Filter by total amount range
 * - Sort by date (ascending/descending)
 * - Pagination
 *
 * @component
 * @selector app-orders-table
 * @standalone true
 */
@Component({
  selector: 'app-orders-table',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './orders-table.component.html',
  styleUrls: ['./orders-table.component.scss'],
})
export class OrdersTableComponent {
  @Input() orderRows: OrderTableRow[] = [];

  showFilter = false;
  currentPage: number = 1;
  itemsPerPage: number = 10;
  sortField: 'date' = 'date';
  sortAsc: boolean = true;

  orderStatus = OrderStatus;

  filter: OrderFilter = {};

  private _searchTerm = '';
  get searchTerm(): string {
    return this._searchTerm;
  }
  set searchTerm(val: string) {
    this._searchTerm = val;
    this.goToPage(1);
  }

  get filteredRows(): OrderTableRow[] {
    let result = this.filterOrderRows(this.orderRows, this.filter);

    result = result.sort((a, b) => {
      const fieldA = a[this.sortField].toLowerCase();
      const fieldB = b[this.sortField].toLowerCase();
      if (fieldA < fieldB) return this.sortAsc ? -1 : 1;
      if (fieldA > fieldB) return this.sortAsc ? 1 : -1;
      return 0;
    });

    return result;
  }

  get paginatedRows(): OrderTableRow[] {
    const start = (this.currentPage - 1) * this.itemsPerPage;
    const end = start + this.itemsPerPage;
    return this.filteredRows.slice(start, end);
  }

  get totalPages(): number {
    return Math.ceil(this.filteredRows.length / this.itemsPerPage);
  }

```

```
  statusClass(status: string) {
    return status;
  }

  openFilter() {
    this.showFilter = !this.showFilter;
  }

  closeFilter() {
    this.showFilter = false;
  }

  apply() {
    this.showFilter = false;

    this.applyFilters();
  }

  applyFilters() {
    this.goToPage(1);
  }

  goToPage(page: number) {
    if (page < 1) page = 1;
    if (page > this.totalPages) page = this.totalPages;
    this.currentPage = page;
  }

  prevPage() {
    this.goToPage(this.currentPage - 1);
  }

  nextPage() {
    this.goToPage(this.currentPage + 1);
  }

  toggleSort() {
    this.sortAsc = !this.sortAsc;
  }

  filterOrderRows(rows: OrderTableRow[], filter: OrderFilter): OrderTableRow[] {
    const lowerSearch = filter.searchTerm?.toLowerCase() ?? '';

    return rows.filter((row) => {
      const matchesSearch =
        lowerSearch === '' ||
        `${row.orderId} ${row.user}`.toLowerCase().includes(lowerSearch);

      const matchesStatus =
        filter.status === undefined ||
        filter.status === null ||
        filter.status === ('' as OrderStatus) ||
        row.status === filter.status;

      const matchesMinTotal =
        filter.minTotal === undefined ||
        filter.minTotal === null ||
        row.total >= filter.minTotal;
      const matchesMaxTotal =
        filter.maxTotal === undefined ||
        filter.maxTotal === null ||
        row.total <= filter.maxTotal;

      if (this.showFilter) {
        return matchesSearch;
      }

      return (
        matchesSearch && matchesStatus && matchesMinTotal && matchesMaxTotal
      );
```

```
    });
  }
}

```

Ø=ÜÁ features\revenue\components\revenue-table

Ø=ÜÄ features\revenue\components\revenue-table\revenue-table.component.ts

```typescript
import { Component, Input, Output, Injectable } from '@angular/core';
import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
import { RevenueFilter, RevenueTableRow } from '../../models/revenue';
import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';

/**
 * Component for displaying revenue data in a table format.
 *
 * This component provides a paginated and filterable table for revenue data.
 * It supports filtering by search term, order count, gross revenue, and total sales,
 * as well as sorting and pagination.
 *
 * Features:
 * - Search by date
 * - Filter by orders, gross revenue, and total sales ranges
 * - Sort by date (ascending/descending)
 * - Pagination
 * - Currency formatting
 *
 * Relations:
 * - NumberFormatterService: Currency formatting
 *
 * @component
 * @selector app-revenue-table
 * @standalone true
 */
@Component({
  selector: 'app-revenue-table',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './revenue-table.component.html',
  styleUrls: ['./revenue-table.component.scss'],
})
@Injectable()
export class RevenueTableComponent {
  @Input() revenueRows: RevenueTableRow[] = [];

  showFilter = false;
  currentPage: number = 1;
  itemsPerPage: number = 10;
  sortField: 'date' = 'date';
  sortAsc: boolean = true;

  filter: RevenueFilter = {};

  private numberFormatter = new NumberFormatterService();

  private _searchTerm = '';
  get searchTerm(): string {
    return this._searchTerm;
  }
  set searchTerm(val: string) {
```

```
    this._searchTerm = val;
    this.goToPage(1);
  }

  get filteredRows(): RevenueTableRow[] {
    let result = this.filterRevenueRows(this.revenueRows, this.filter);

    result = result.sort((a, b) => {
      const fieldA = a[this.sortField].toLowerCase();
      const fieldB = b[this.sortField].toLowerCase();
      if (fieldA < fieldB) return this.sortAsc ? -1 : 1;
      if (fieldA > fieldB) return this.sortAsc ? 1 : -1;
      return 0;
    });

    return result;
  }

  get paginatedRows(): RevenueTableRow[] {
    const start = (this.currentPage - 1) * this.itemsPerPage;
    const end = start + this.itemsPerPage;
    return this.filteredRows.slice(start, end);
  }

  get totalPages(): number {
    return Math.ceil(this.filteredRows.length / this.itemsPerPage);
  }

  openFilter() {
    this.showFilter = !this.showFilter;
  }

  closeFilter() {
    this.showFilter = false;
  }

  apply() {
    this.showFilter = false;

    this.applyFilters();
  }

  applyFilters() {
    this.goToPage(1);
  }

  goToPage(page: number) {
    if (page < 1) page = 1;
    if (page > this.totalPages) page = this.totalPages;
    this.currentPage = page;
  }

  prevPage() {
    this.goToPage(this.currentPage - 1);
  }

  nextPage() {
    this.goToPage(this.currentPage + 1);
  }

  toggleSort() {
    this.sortAsc = !this.sortAsc;
  }

  filterRevenueRows(
    rows: RevenueTableRow[],
    filter: RevenueFilter
  ): RevenueTableRow[] {
    const lowerSearch = filter.searchTerm?.toLowerCase() ?? '';

```

```
    return rows.filter((row) => {
      const matchesSearch =
        lowerSearch === '' || row.date.toLowerCase().includes(lowerSearch);

      const matchesMinOrders =
        filter.minOrders === undefined ||
        filter.minOrders === null ||
        row.orders >= filter.minOrders;
      const matchesMaxOrders =
        filter.maxOrders === undefined ||
        filter.maxOrders === null ||
        row.orders <= filter.maxOrders;

      const matchesMinGrossRevenue =
        filter.minGrossRevenue === undefined ||
        filter.minGrossRevenue === null ||
        row.grossRevenue >= filter.minGrossRevenue;
      const matchesMaxGrossRevenue =
        filter.maxGrossRevenue === undefined ||
        filter.maxGrossRevenue === null ||
        row.grossRevenue <= filter.maxGrossRevenue;

      const matchesMinTotalSales =
        filter.minTotalSales === undefined ||
        filter.minTotalSales === null ||
        row.totalSales >= filter.minTotalSales;
      const matchesMaxTotalSales =
        filter.maxTotalSales === undefined ||
        filter.maxTotalSales === null ||
        row.totalSales <= filter.maxTotalSales;

      if (this.showFilter) {
        return matchesSearch;
      }

      return (
        matchesSearch &&
        matchesMinOrders &&
        matchesMaxOrders &&
        matchesMinGrossRevenue &&
        matchesMaxGrossRevenue &&
        matchesMinTotalSales &&
        matchesMaxTotalSales
      );
    });
  }

  formatCurrency(value: number | null | undefined): string {
    return this.numberFormatter.formatCurrency(value);
  }
}

```

Ø=ÜÁ features\revenue\components\revenueGraph

Ø=ÜÄ features\revenue\components\revenueGraph\revenue-graph.component.ts

```
import { CommonModule } from '@angular/common';
import { Component, Inject, Input, PLATFORM_ID } from '@angular/core';
import { NgApexchartsModule } from 'ng-apexcharts';
import { FormsModule } from '@angular/forms';
import { DailyRevenueData, MonthlyRevenueData } from '../../models/revenue';
import { NumberFormatterService } from '../../../../shared/utils/number-formatter.service';
```

```

/**
 * Component for displaying revenue charts.
 *
 * This component displays revenue data in either bar or area chart format,
 * with support for filtering by day, month, or year. It uses ApexCharts
 * for visualization and includes interactive tooltips showing percentage changes.
 *
 * Features:
 * - Bar and area chart types
 * - Filter by day, month, or year
 * - Interactive tooltips with percentage changes
 * - Automatic data sorting and formatting
 * - Gross revenue calculation
 *
 * Relations:
 * - NgApexchartsModule: Chart rendering
 * - NumberFormatterService: Value formatting
 *
 * @component
 * @selector app-revenue-graph
 * @standalone true
 */
@Component({
  selector: 'app-revenue-graph',
  templateUrl: './revenue-graph.component.html',
  styleUrls: ['./revenue-graph.component.scss'],
  standalone: true,
  imports: [CommonModule, NgApexchartsModule, FormsModule],
})
export class RevenueGraphComponent {
  @Input() dailyData!: DailyRevenueData[];
  @Input() monthlyData!: MonthlyRevenueData[];

  actualYear: number = new Date().getFullYear();
  private numberFormatter = new NumberFormatterService();

  filterType: 'day' | 'month' | 'year' = 'day';

  chartType: 'bar' | 'area' = 'bar';

  public chartOptions: any;

  constructor(@Inject(PLATFORM_ID) private platformId: any) {}

  ngOnInit() {}

  get filterTypeLabel(): string {
    switch (this.filterType) {
      case 'day':
        return 'By day';
      case 'month':
        return 'By month';
      case 'year':
        return 'By year';
    }
  }

  setChartType(type: 'bar' | 'area') {
    this.chartType = type;
    this.changeChartData();
  }

  setFilterType(type: 'day' | 'month' | 'year') {
    this.filterType = type;
    this.changeChartData();
  }

  onFilterTypeChange(event: any) {
    this.filterType = event.target.value;
```

```
    this.changeChartData();
  }

  ngOnChanges() {
    if (this.dailyData) {
      this.chartOptions = this.getChartOptions(this.dailyData, this.chartType);
    }
  }

  changeChartData() {
    if (this.filterType === 'day') {
      this.chartOptions = this.getChartOptions(this.dailyData, this.chartType);
    } else if (this.filterType === 'month') {
      this.chartOptions = this.getChartOptions(
        this.monthlyData,
        this.chartType
      );
    }
  }

  getChartOptions(
    data: DailyRevenueData[] | MonthlyRevenueData[],
    chartType: 'bar' | 'area'
  ): any {
    const isDailyData = 'date' in data[0];
    let categories: string[] = [];

    if (isDailyData) {
      categories = data
        .map((d) => new Date((d as DailyRevenueData).date).getDate().toString())
        .sort((a, b) => parseInt(a) - parseInt(b));
    } else {
      categories = (data as MonthlyRevenueData[]).map((d) => {
        const dateObj = new Date(d.year, d.month - 1);
        return dateObj.toLocaleString('en-US', {
          month: 'short',
          year: 'numeric',
        });
      });
    }

    let sortedData = data.slice();
    if (isDailyData) {
      sortedData = sortedData.sort(
        (a, b) =>
          new Date((a as DailyRevenueData).date).getDate() -
          new Date((b as DailyRevenueData).date).getDate()
      );
    } else {
      sortedData = sortedData.sort((a, b) =>
        (a as MonthlyRevenueData).year === (b as MonthlyRevenueData).year
          ? (a as MonthlyRevenueData).month - (b as MonthlyRevenueData).month
          : (a as MonthlyRevenueData).year - (b as MonthlyRevenueData).year
      );
    }

    const chartData = sortedData.map((d) => d.grossRevenue);
    const actualYear = this.actualYear;

    if (chartType === 'area') {
      return {
        chart: {
          type: 'area',
          height: 350,
          toolbar: { show: false },
          foreColor: '#fff',
          fontFamily: 'Inter, Arial, sans-serif',
          background: 'transparent',
        },
        series: [
```

```
          {
            name: 'months',
            data: chartData,
          },
        ],
        xaxis: {
          categories: categories,
          labels: {
            style: { colors: '#d8d8d8' },
          },
          axisBorder: { show: false },
          axisTicks: { show: false },
        },
        yaxis: {
          labels: {
            style: { colors: '#d8d8d8' },
          },
        },
        grid: {
          borderColor: '#333',
          strokeDashArray: 4,
          xaxis: {
            lines: {
              show: true,
            },
          },
        },
        dataLabels: { enabled: false },
        stroke: {
          curve: 'straight',
          width: 1,
          colors: ['#EAF2F8'],
        },
        fill: {
          type: 'gradient',
          gradient: {
            shade: 'dark',
            type: 'vertical',
            gradientToColors: ['#3967D7'],
            opacityFrom: 0.9,
            opacityTo: 0,
          },
        },
        tooltip: {
          theme: 'dark',
          x: { show: true },
          custom: function ({ series, seriesIndex, dataPointIndex, w }: any) {
            const value = series[seriesIndex][dataPointIndex];

            const prevValue =
              dataPointIndex > 0
                ? series[seriesIndex][dataPointIndex - 1]
                : null;
            let percentDiff: number | null = 0;
            let direction = null;
            let cardClass = '';
            let validatorClass = '';

            if (prevValue !== null && prevValue !== 0) {
              percentDiff = Math.round(
                ((value - prevValue) / Math.abs(prevValue)) * 100
              );
              direction = percentDiff > 0 ? 'up' : 'down';
              if (direction === 'up') {
                cardClass = 'positive-container';
                validatorClass = 'positive-validator';
              } else {
                cardClass = 'negative-container';
                validatorClass = 'negative-validator';
              }
```

```
            } else {
              percentDiff = null;
              direction = null;
            }

            return `<div class=" ${cardClass} regularText color-background d-flex flex-col
toolTip-containers items-start ">
          <p class="smallText color-text-gray">${actualYear}</p>
          <div class="d-flex text-container items-center ">
                    <p class= "subtitle">
                    $${value}
          </p>
          ${
            percentDiff != null
              ? `<span class="smallText py-4 px-6 d-flex justify-center items-center
${validatorClass}"> ${percentDiff}% <span class="${
                    direction === 'up'
                    ? 'icon-status-arrow-up'
                    : 'icon-status-arrow'
                } ml-3"></span></span>`
              : ''
          }
          </div>


          </div>`;
          },
          position: function (data: any, opts: any) {
            return {
              left: data.point.x,
              top: data.point.y - 160,
            };
          },
        },
      };
    }

    return {
      chart: {
        type: 'bar',
        height: 350,
        toolbar: { show: false },
        foreColor: '#fff',
        background: 'transparent',
        fontFamily: 'Inter, Arial, sans-serif',
      },
      series: [
        {
          name: 'Gross Revenue',
          data: chartData,
        },
      ],
      fill: {
        colors: ['#F44336', '#E91E63', '#9C27B0'],
      },

      xaxis: {
        categories: categories,
        labels: { style: { colors: '#d8d8d8' } },
        axisBorder: { show: false },
        axisTicks: { show: false },
      },
      yaxis: {
        labels: { style: { colors: '#d8d8d8' } },
      },
      grid: {
        borderColor: '#333',
        strokeDashArray: 4,
        xaxis: { lines: { show: true } },
      },
      dataLabels: { enabled: false },
      tooltip: {
```

```
        theme: 'dark',
        x: { show: true },
        custom: ({ series, seriesIndex, dataPointIndex, w }: any) => {
          const value = series[seriesIndex][dataPointIndex];
          const prevValue =
            dataPointIndex > 0 ? series[seriesIndex][dataPointIndex - 1] : null;
          let percentDiff: number | null = 0;
          let direction = null;
          let cardClass = '';
          let validatorClass = '';

          if (prevValue !== null && prevValue !== 0) {
            percentDiff = Math.round(
              ((value - prevValue) / Math.abs(prevValue)) * 100
            );
            direction = percentDiff > 0 ? 'up' : 'down';
            if (direction === 'up') {
              cardClass = 'positive-container';
              validatorClass = 'positive-validator';
            } else {
              cardClass = 'negative-container';
              validatorClass = 'negative-validator';
            }
          } else {
            percentDiff = null;
            direction = null;
          }

          const formattedValue = this.numberFormatter.formatCurrency(value);
          const formattedPercent = this.numberFormatter.formatPercentageValue(percentDiff);

          return `<div class=" ${cardClass} regularText color-background d-flex flex-col
toolTip-container items-start ">
            <p class= "smallText color-text-gray">${actualYear}</p>
            <div class="d-flex text-container items-center ">
              <p class= "subtitle">${formattedValue}</p>
              ${
                percentDiff != null
                  ? `<span class="smallText py-4 px-6 d-flex justify-center items-center
${validatorClass}">  ${formattedPercent}% <span class="${
                      direction === 'up'
                        ? 'icon-status-arrow-up'
                        : 'icon-status-arrow'
                    } ml-3"></span>
                     </span>`
                  : ''
              }
            </div>
          </div>`;
        },
      },
    };
  }

  capitalizeFirstLetter(str: string): string {
    if (!str) return '';
    return str.charAt(0).toUpperCase() + str.slice(1);
  }

  get grossRevenue(): number {
    const result = this.dailyData.reduce(
      (acc, curr) => acc + (curr.grossRevenue ?? 0),
      0
    );
    return result;
  }
}
```

Ø=ÜÁ features\revenue\components\subscriptions-table

Ø=ÜÄ features\revenue\components\subscriptions-table\subscriptions-table.component.ts

```typescript
import { Component, Input, Output } from '@angular/core';
import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
import {
  SubscriptionFilter,
  SubscriptionStatus,
  SubscriptionTableRow,
} from '../../models/revenue';

/**
 * Component for displaying subscriptions data in a table format.
 *
 * This component provides a paginated and filterable table for subscription data.
 * It supports filtering by search term, subscription status, and total amount,
 * as well as sorting and pagination.
 *
 * Features:
 * - Search by subscription name
 * - Filter by subscription status (Active, Pending, Failed)
 * - Filter by total amount range (parses currency strings)
 * - Sort by start date (ascending/descending)
 * - Pagination
 *
 * @component
 * @selector app-subscriptions-table
 * @standalone true
 */
@Component({
  selector: 'app-subscriptions-table',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './subscriptions-table.component.html',
  styleUrls: ['./subscriptions-table.component.scss'],
})
export class SubscriptionsTableComponent {
  @Input() subscriptionRows: SubscriptionTableRow[] = [];

  showFilter = false;
  currentPage: number = 1;
  itemsPerPage: number = 10;
  sortField: 'startDate' = 'startDate';
  sortAsc: boolean = true;

  subscriptionStatus = SubscriptionStatus;

  filter: SubscriptionFilter = {};

  private _searchTerm = '';
  get searchTerm(): string {
    return this._searchTerm;
  }
  set searchTerm(val: string) {
    this._searchTerm = val;
    this.goToPage(1);
  }

  get filteredRows(): SubscriptionTableRow[] {
    let result = this.filterOrderRows(this.subscriptionRows, this.filter);

    result = result.sort((a, b) => {
```

```
      const fieldA = a[this.sortField].toLowerCase();
      const fieldB = b[this.sortField].toLowerCase();
      if (fieldA < fieldB) return this.sortAsc ? -1 : 1;
      if (fieldA > fieldB) return this.sortAsc ? 1 : -1;
      return 0;
    });

    return result;
  }

  get paginatedRows(): SubscriptionTableRow[] {
    const start = (this.currentPage - 1) * this.itemsPerPage;
    const end = start + this.itemsPerPage;
    return this.filteredRows.slice(start, end);
  }

  get totalPages(): number {
    return Math.ceil(this.filteredRows.length / this.itemsPerPage);
  }

  statusClass(status: string) {
    return status;
  }

  openFilter() {
    this.showFilter = !this.showFilter;
  }

  closeFilter() {
    this.showFilter = false;
  }

  apply() {
    this.showFilter = false;

    this.applyFilters();
  }

  applyFilters() {
    this.goToPage(1);
  }

  goToPage(page: number) {
    if (page < 1) page = 1;
    if (page > this.totalPages) page = this.totalPages;
    this.currentPage = page;
  }

  prevPage() {
    this.goToPage(this.currentPage - 1);
  }

  nextPage() {
    this.goToPage(this.currentPage + 1);
  }

  toggleSort() {
    this.sortAsc = !this.sortAsc;
  }

  filterOrderRows(
    rows: SubscriptionTableRow[],
    filter: SubscriptionFilter
  ): SubscriptionTableRow[] {
    const lowerSearch = filter.searchTerm?.toLowerCase() ?? '';

    return rows.filter((row) => {
      const matchesSearch =
        lowerSearch === '' ||
        row.subscription.toLowerCase().includes(lowerSearch);
```

```
132
133       const matchesStatus =
134         filter.status === undefined ||
135         filter.status === null ||
136         filter.status === ('' as SubscriptionStatus) ||
137         row.status === filter.status;
138
139       const matchesMinTotal =
140         filter.minTotal === undefined ||
141         filter.minTotal === null ||
142         parseFloat(row.total.replace('$', '').split('/')[0]) >= filter.minTotal;
143
144       const matchesMaxTotal =
145         filter.maxTotal === undefined ||
146         filter.maxTotal === null ||
147         parseFloat(row.total.replace('$', '').split('/')[0]) <= filter.maxTotal;
148
149       if (this.showFilter) {
150         return matchesSearch;
151       }
152
153       return (
154         matchesSearch && matchesStatus && matchesMinTotal && matchesMaxTotal
155       );
156     });
157   }
158 }
159
```

Ø=ÜÁ features\revenue\mocks

Ø=ÜÄ features\revenue\mocks\revenue_mock.ts

```
1  import {
2    DailyRevenueData,
3    MonthlyRevenueData,
4    OrderStatus,
5    OrderTableRow,
6    RevenueSummary,
7    RevenueTableRow,
8    SubscriptionStatus,
9    SubscriptionTableRow,
10   YearlyRevenueData,
11 } from '../models/revenue';
12
13 /**
14  * Mock data for revenue summary.
15  *
16  * Used for development and testing purposes. Contains aggregated revenue metrics.
17  *
18  * @constant mockRevenueSummary
19  * @type {RevenueSummary}
20  */
21 export const mockRevenueSummary: RevenueSummary = {
22   grossRevenue: 45750,
23   returns: 56,
24   coupons: 1870,
25   netRevenue: 24780,
26   totalRevenue: 24780,
27 };
28
29 /**
30  * Mock data for daily revenue.
```

```
 *
 * Contains daily revenue data for a month (August 2025).
 * Used for chart visualization in development.
 *
 * @constant dailyRevenueMock
 * @type {DailyRevenueData[]}
 */
export const dailyRevenueMock: DailyRevenueData[] = [
  { date: '2025-08-01', grossRevenue: 400 },
  { date: '2025-08-02', grossRevenue: 700 },
  { date: '2025-08-03', grossRevenue: 1100 },
  { date: '2025-08-04', grossRevenue: 1700 },
  { date: '2025-08-05', grossRevenue: 3000 },
  { date: '2025-08-06', grossRevenue: 2350 },
  { date: '2025-08-07', grossRevenue: 900 },
  { date: '2025-08-08', grossRevenue: 1200 },
  { date: '2025-08-09', grossRevenue: 1500 },
  { date: '2025-08-10', grossRevenue: 1800 },
  { date: '2025-08-11', grossRevenue: 400 },
  { date: '2025-08-12', grossRevenue: 700 },
  { date: '2025-08-13', grossRevenue: 1100 },
  { date: '2025-08-14', grossRevenue: 1700 },
  { date: '2025-08-15', grossRevenue: 3000 },
  { date: '2025-08-16', grossRevenue: 2350 },
  { date: '2025-08-17', grossRevenue: 900 },
  { date: '2025-08-18', grossRevenue: 1200 },
  { date: '2025-08-19', grossRevenue: 1500 },
  { date: '2025-08-20', grossRevenue: 1800 },
  { date: '2025-08-21', grossRevenue: 400 },
  { date: '2025-08-22', grossRevenue: 700 },
  { date: '2025-08-23', grossRevenue: 1100 },
  { date: '2025-08-24', grossRevenue: 1700 },
  { date: '2025-08-25', grossRevenue: 3000 },
  { date: '2025-08-26', grossRevenue: 2350 },
  { date: '2025-08-27', grossRevenue: 900 },
  { date: '2025-08-28', grossRevenue: 1200 },
  { date: '2025-08-29', grossRevenue: 1500 },
  { date: '2025-08-30', grossRevenue: 1800 },
  { date: '2025-08-31', grossRevenue: 1800 },
];

/**
 * Mock data for monthly revenue.
 *
 * Contains monthly revenue data for the year 2025.
 * Used for chart visualization in development.
 *
 * @constant monthlyRevenueMock
 * @type {MonthlyRevenueData[]}
 */
export const monthlyRevenueMock: MonthlyRevenueData[] = [
  { year: 2025, month: 1, grossRevenue: 21000 },
  { year: 2025, month: 2, grossRevenue: 19800 },
  { year: 2025, month: 3, grossRevenue: 23050 },
  { year: 2025, month: 4, grossRevenue: 25500 },
  { year: 2025, month: 5, grossRevenue: 24800 },
  { year: 2025, month: 6, grossRevenue: 26706.52 },
  { year: 2025, month: 7, grossRevenue: 28000 },
  { year: 2025, month: 8, grossRevenue: 29000 },
  { year: 2025, month: 9, grossRevenue: 30000 },
  { year: 2025, month: 10, grossRevenue: 31000 },
  { year: 2025, month: 11, grossRevenue: 32000 },
  { year: 2025, month: 12, grossRevenue: 33000 },
];

/**
 * Mock data for revenue table.
 *
 * Contains sample revenue table rows with daily revenue breakdown.
 * Used for table display in development.
```

```
 *
 * @constant revenueTableMock
 * @type {RevenueTableRow[]}
 */
export const revenueTableMock: RevenueTableRow[] = [
  {
    date: 'June 23,2025',
    orders: 2,
    grossRevenue: 665.0,
    returns: 0.0,
    coupons: 52.0,
    netSales: 603.0,
    taxes: 0.0,
    shipping: 0.0,
    totalSales: 603.0,
  },
  {
    date: 'June 22,2025',
    orders: 2,
    grossRevenue: 377.5,
    returns: 0.0,
    coupons: 0.0,
    netSales: 377.5,
    taxes: 0.0,
    shipping: 0.0,
    totalSales: 377.5,
  },
  {
    date: 'June 21,2025',
    orders: 0,
    grossRevenue: 0.0,
    returns: 0.0,
    coupons: 0.0,
    netSales: 0.0,
    taxes: 0.0,
    shipping: 0.0,
    totalSales: 0.0,
  },
  {
    date: 'June 20,2025',
    orders: 3,
    grossRevenue: 189.0,
    returns: 0.0,
    coupons: 7.0,
    netSales: 182.0,
    taxes: 0.0,
    shipping: 0.0,
    totalSales: 182.0,
  },
  {
    date: 'June 19,2025',
    orders: 0,
    grossRevenue: 0.0,
    returns: 0.0,
    coupons: 0.0,
    netSales: 0.0,
    taxes: 0.0,
    shipping: 0.0,
    totalSales: 0.0,
  },
];

/**
 * Mock data for orders table.
 *
 * Contains sample order rows with order details.
 * Used for orders table display in development.
 *
 * @constant orderTableMock
 * @type {OrderTableRow[]}
```

```
 */
export const orderTableMock: OrderTableRow[] = [
  {
    orderId: '#3686',
    user: 'Andrés Valdes',
    date: '3 hours ago',
    status: OrderStatus.Completed,
    total: 127.0,
    affiliateReferral: '#532',
    origin: 'Referral:Tx3funding.com',
  },
  {
    orderId: '#3686',
    user: 'Marvin McKinney',
    date: '3 hours ago',
    status: OrderStatus.Pending,
    total: 127.0,
    affiliateReferral: null,
    origin: 'Referral:Tx3funding.com',
  },
  {
    orderId: '#3686',
    user: 'Guy Hawkins',
    date: '5 hours ago',
    status: OrderStatus.Completed,
    total: 127.0,
    affiliateReferral: '#532',
    origin: 'Referral:Tx3funding.com',
  },
  {
    orderId: '#3686',
    user: 'Ralph Edwards',
    date: '6 hours ago',
    status: OrderStatus.Cancelled,
    total: 127.0,
    affiliateReferral: '#532',
    origin: 'Referral:Tx3funding.com',
  },
  {
    orderId: '#3686',
    user: 'Jenny Wilson',
    date: '6 hours ago',
    status: OrderStatus.Failed,
    total: 127.0,
    affiliateReferral: '#532',
    origin: 'Referral:Tx3funding.com',
  },
];

/**
 * Mock data for subscriptions table.
 *
 * Contains sample subscription rows with subscription details.
 * Used for subscriptions table display in development.
 *
 * @constant subscriptionTableMock
 * @type {SubscriptionTableRow[]}
 */
export const subscriptionTableMock: SubscriptionTableRow[] = [
  {
    status: SubscriptionStatus.Active,
    subscription: '#3686 Andrés Valdes',
    items: 'Pro Model',
    total: '$157.00/month',
    startDate: '3 hours ago',
    trialEnd: '-',
    nextPayment: 'July 23, 2025',
    lastOrderDate: '-',
    endDate: '-',
    orders: 1,
```

```
  },
  {
    status: SubscriptionStatus.Pending,
    subscription: '#3686 Andrés Valdes',
    items: 'Pro Model',
    total: '$157.00/month',
    startDate: '3 hours ago',
    trialEnd: '-',
    nextPayment: 'July 23, 2025',
    lastOrderDate: '-',
    endDate: '-',
    orders: 1,
  },
  {
    status: SubscriptionStatus.Active,
    subscription: '#3686 Andrés Valdes',
    items: 'Pro Model',
    total: '$157.00/month',
    startDate: '3 hours ago',
    trialEnd: '-',
    nextPayment: 'July 23, 2025',
    lastOrderDate: '-',
    endDate: '-',
    orders: 1,
  },
  {
    status: SubscriptionStatus.Failed,
    subscription: '#3686 Andrés Valdes',
    items: 'Pro Model',
    total: '$157.00/month',
    startDate: '3 hours ago',
    trialEnd: '-',
    nextPayment: 'July 23, 2025',
    lastOrderDate: '-',
    endDate: '-',
    orders: 1,
  },
  {
    status: SubscriptionStatus.Active,
    subscription: '#3686 Andrés Valdes',
    items: 'Pro Model',
    total: '$157.00/month',
    startDate: '3 hours ago',
    trialEnd: '-',
    nextPayment: 'July 23, 2025',
    lastOrderDate: '-',
    endDate: '-',
    orders: 1,
  },
];

```

Ø=ÜÁ features\revenue\models

Ø=ÜÄ features\revenue\models\revenue.ts

```
/**
 * Enum representing the possible statuses of an order.
 *
 * @enum {string}
 */
export enum OrderStatus {
  Completed = 'Completed',
```

```
  Pending = 'Pending',
  Cancelled = 'Cancelled',
  Failed = 'Failed',
}

/**
 * Enum representing the possible statuses of a subscription.
 *
 * @enum {string}
 */
export enum SubscriptionStatus {
  Active = 'Active',
  Pending = 'Pending',
  Failed = 'Failed',
}

/**
 * Interface representing a summary of revenue data.
 *
 * Contains aggregated revenue metrics including gross revenue, returns, coupons,
 * net revenue, and total revenue.
 *
 * @interface RevenueSummary
 */
export interface RevenueSummary {
  grossRevenue: number;
  returns: number;
  coupons: number;
  netRevenue: number;
  totalRevenue: number;
}

/**
 * Interface representing a row in the revenue table.
 *
 * Contains daily revenue data including orders count, gross revenue, returns,
 * coupons, net sales, taxes, shipping, and total sales.
 *
 * @interface RevenueTableRow
 */
export interface RevenueTableRow {
  date: string;
  orders: number;
  grossRevenue: number;
  returns: number;
  coupons: number;
  netSales: number;
  taxes: number;
  shipping: number;
  totalSales: number;
}

/**
 * Interface representing daily revenue data.
 *
 * Used for chart visualization showing revenue trends by day.
 *
 * @interface DailyRevenueData
 */
export interface DailyRevenueData {
  date: string;
  grossRevenue: number;
}

/**
 * Interface representing monthly revenue data.
 *
 * Used for chart visualization showing revenue trends by month.
 *
 * @interface MonthlyRevenueData
```

```ts
 */
export interface MonthlyRevenueData {
  year: number;
  month: number;
  grossRevenue: number;
}

/**
 * Interface representing yearly revenue data.
 *
 * Used for chart visualization showing revenue trends by year.
 *
 * @interface YearlyRevenueData
 */
export interface YearlyRevenueData {
  year: number;
  grossRevenue: number;
}

/**
 * Interface representing a row in the orders table.
 *
 * Contains order information including order ID, user, date, status, total,
 * affiliate referral, and origin.
 *
 * @interface OrderTableRow
 */
export interface OrderTableRow {
  orderId: string;
  user: string;
  date: string;
  status: OrderStatus;
  total: number;
  affiliateReferral: string | null;
  origin: string;
}

/**
 * Interface representing a row in the subscriptions table.
 *
 * Contains subscription information including status, subscription details,
 * items, total, dates, and order count.
 *
 * @interface SubscriptionTableRow
 */
export interface SubscriptionTableRow {
  status: SubscriptionStatus;
  subscription: string;
  items: string;
  total: string;
  startDate: string;
  trialEnd: string;
  nextPayment: string;
  lastOrderDate: string;
  endDate: string;
  orders: number;
}

/**
 * Interface representing filter criteria for revenue table.
 *
 * Used to filter revenue data by search term, order count, gross revenue, and total sales.
 *
 * @interface RevenueFilter
 */
export interface RevenueFilter {
  searchTerm?: string;
  minOrders?: number;
  maxOrders?: number;
  minGrossRevenue?: number;
```

```
  maxGrossRevenue?: number;
  minTotalSales?: number;
  maxTotalSales?: number;
}

/**
 * Interface representing filter criteria for orders table.
 *
 * Used to filter orders by search term, status, and total amount.
 *
 * @interface OrderFilter
 */
export interface OrderFilter {
  searchTerm?: string;
  status?: OrderStatus;
  minTotal?: number;
  maxTotal?: number;
}

/**
 * Interface representing filter criteria for subscriptions table.
 *
 * Used to filter subscriptions by search term, status, and total amount.
 *
 * @interface SubscriptionFilter
 */
export interface SubscriptionFilter {
  searchTerm?: string;
  status?: SubscriptionStatus;
  minTotal?: number;
  maxTotal?: number;
}

```

Ø=ÜÁ features\strategy

Ø=ÜÄ features\strategy\strategy.component.ts

```
import { CommonModule } from '@angular/common';
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Router } from '@angular/router';
import { FormsModule } from '@angular/forms';
import { User } from '../overview/models/overview';
import { selectUser } from '../auth/store/user.selectios';
import { SettingsService } from './service/strategy.service';
import { ConfigurationOverview } from './models/strategy.model';
import { TextInputComponent, StrategyCardComponent, StrategyCardData,
StrategyGridModalComponent, LoadingSpinnerComponent, PlanBannerComponent } from '../../
shared/components';
import { ConfirmPopupComponent } from '../../shared/pop-ups/confirm-pop-up/confirm-
import { Store } from '@ngrx/store';
import { ReportService } from '../report/service/report.service';
import { AuthService } from '../auth/service/authService';
import { initialStrategyState } from '../strategy/store/strategy.reducer';
import { selectTradeWin } from '../report/store/report.selectors';
import { selectTotalTrades } from '../report/store/report.selectors';
import { resetConfig } from '../strategy/store/strategy.actions';
import { StrategyState } from '../strategy/models/strategy.model';
import { AccountData } from '../auth/models/userModel';
import { setUserKey } from '../report/store/report.actions';
import { selectUserKey } from '../report/store/report.selectors';
import { GlobalStrategyUpdaterService } from '../../shared/services/global-strategy-
updater.service';
import { attRates } from '../strategy/store/strategy.selectors';
import { selectNetPnL } from '../report/store/report.selectors';
import { PlanLimitationsGuard } from '../../guards/plan-limitations.guard';
```

```typescript
import { AppContextService } from '../../shared/context';
import { StrategyCacheService } from './services/strategy-cache.service';
import { BalanceCacheService } from './services/balance-cache.service';
import { AlertService } from '../../shared/services/alert.service';


/**
 * Main component for managing trading strategies.
 *
 * This component provides comprehensive strategy management functionality including:
 * - Creating, editing, activating, copying, and deleting strategies
 * - Displaying strategy cards with statistics
 * - Managing multiple strategies per user
 * - Plan limitation detection and enforcement
 * - Strategy guide modal for first-time users
 * - Caching strategies for performance
 * - Real-time balance updates for risk calculations
 *
 * Key Features:
 * - Multiple strategy support with active/inactive states
 * - Strategy caching for fast access
 * - Balance caching for risk calculations
 * - Plan limitation checks (max strategies per plan)
 * - Strategy guide for new users
 * - Search and filter strategies
 * - Strategy cards with win rate, active rules, and days active
 *
 * Data Flow:
 * 1. Component initializes and loads user data
 * 2. Loads all user accounts
 * 3. Loads all strategies (overview + configuration) into cache
 * 4. Sets up listeners for context data and report data
 * 5. Updates strategy cards with real-time data
 *
 * Relations:
 * - SettingsService: CRUD operations for strategies
 * - StrategyCacheService: Caching strategy data
 * - BalanceCacheService: Caching account balances
 * - PlanLimitationsGuard: Checking plan limitations
 * - AppContextService: Global state management
 * - ReportService: Fetching account balances
 * - Store (NgRx): Local state for strategy rules
 * - EditStrategyComponent: Editing strategy details
 * - StrategyCardComponent: Displaying strategy cards
 *
 * @component
 * @selector app-strategy
 * @standalone true
 */
@Component({
  selector: 'app-strategy',
  imports: [
    CommonModule,
    FormsModule,
    TextInputComponent,
    StrategyCardComponent,
    StrategyGuideModalComponent,
    LoadingSpinnerComponent,
    PlanBannerComponent,
    ConfirmPopupComponent,
  ],
  templateUrl: './strategy.component.html',
  styleUrl: './strategy.component.scss',
  standalone: true,
})
export class Strategy implements OnInit, OnDestroy {
  config: any = {};
  user: User | null = null;
  loading = false;
  initialLoading = true;
```

```
96
97    // Plan detection and banner
98    accountsData: AccountData[] = [];
99    showPlanBanner = false;
100   planBannerMessage = '';
101   planBannerType = 'info'; // 'info', 'warning', 'success'
102
103
104   // Button state
105   isAddStrategyDisabled = false;
106
107   // Strategy guide modal
108   showStrategyGuide = false;
109
110   // Loading state for strategy creation
111   isCreatingStrategy = false;
112
113   // Loading state for strategy operations (activate, delete, copy)
114   isProcessingStrategy = false;
115
116   // Delete strategy confirmation popup
117   showDeleteConfirmPopup = false;
118   strategyToDeleteId: string = '';
119
120   // Report data for strategy card
121   tradeWin: number = 0;
122   totalTrades: number = 0;
123   netPnL: number = 0;
124
125   // Nuevas propiedades para múltiples estrategias
126   userStrategies: ConfigurationOverview[] = [];
127   activeStrategy: ConfigurationOverview | null = null;
128   filteredStrategies: ConfigurationOverview[] = [];
129   strategyCardsData: StrategyCardData[] = [];
130   searchTerm = '';
131
132   // Strategy Card Data - Dynamic
133   strategyCard: StrategyCardData = {
134     id: '1',
135     name: 'My Trading Strategy',
136     status: true,
137     lastModified: 'Never',
138     rules: 0,
139     days_active: 0,
140     winRate: 0,
141     isFavorite: false,
142     created_at: null,
143     updated_at: null,
144     userId: '',
145     configurationId: ''
146   };
147
148   // CACHE CENTRALIZADO - Usar servicio de cache
149
150   constructor(
151     private store: Store,
152     private router: Router,
153     private strategySvc: SettingsService,
154     private reportSvc: ReportService,
155     private authService: AuthService,
156     private globalStrategyUpdater: GlobalStrategyUpdaterService,
157     private planLimitationsGuard: PlanLimitationsGuard,
158     private appContext: AppContextService,
159     private strategyCacheService: StrategyCacheService,
160     private balanceCacheService: BalanceCacheService,
161     private alertService: AlertService
162   ) {}
163
164   async ngOnInit(): Promise<void> {
165     this.initialLoading = true;
```

```
166
167       try {
168         // FLUJO SIMPLIFICADO: Una sola secuencia de inicialización
169         await this.initializeEverything();
170       } catch (error) {
171         console.error('Error during initialization:', error);
172       } finally {
173         this.initialLoading = false;
174       }
175     }
176
177     /**
178      * FLUJO SIMPLIFICADO DE INICIALIZACIÓN
179      * 1. Obtener usuario
180      * 2. Cargar cuentas
181      * 3. Cargar TODAS las estrategias completas (overview + configuration) en cache
182      * 4. Configurar listeners
183      * 5. Cargar datos de reporte
184      */
185     private async initializeEverything(): Promise<void> {
186       // 1. Obtener usuario
187       await this.initializeUserData();
188
189       if (!this.user?.id) {
190         console.error('No user ID available');
191         return;
192       }
193
194       // 2. Cargar cuentas
195       await this.initializeAccounts();
196
197       // 3. Cargar TODAS las estrategias completas en cache
198       await this.loadAllStrategiesToCache();
199
200       // 4. Configurar listeners
201       this.setupListeners();
202
203       // 5. Cargar datos de reporte
204       await this.initializeReportData();
205     }
206
207     private subscribeToContextData() {
208       // Suscribirse a los datos del usuario
209       this.appContext.currentUser$.subscribe(user => {
210         this.user = user;
211       });
212
213       // Suscribirse a las cuentas del usuario
214       this.appContext.userAccounts$.subscribe(accounts => {
215         this.accountsData = accounts;
216       });
217
218       // Suscribirse a las estrategias del usuario
219       this.appContext.userStrategies$.subscribe(strategies => {
220         this.userStrategies = strategies;
221         this.filteredStrategies = strategies;
222         this.updateStrategyCard();
223       });
224
225       // Suscribirse a los estados de carga
226       this.appContext.isLoading$.subscribe(loading => {
227         this.loading = loading.strategies;
228       });
229
230       // Suscribirse a los errores
231       this.appContext.errors$.subscribe(errors => {
232         if (errors.strategies) {
233           console.error('Error en estrategias:', errors.strategies);
234         }
235       });
```

```
  }

  /**
   * Inicializa los datos del usuario
   */
  private async initializeUserData(): Promise<void> {
    return new Promise((resolve) => {
      this.store.select(selectUser).subscribe({
        next: (user) => {
          this.user = user.user;
          resolve();
        },
        error: (err) => {
          console.error('Error fetching user data', err);
          resolve();
        },
      });
    });
  }

  /**
   * Inicializa las cuentas del usuario
   */
  private async initializeAccounts(): Promise<void> {
    if (this.user?.id) {
      try {
        const accounts = await this.authService.getUserAccounts(this.user.id);
        this.accountsData = accounts || [];
        await this.checkPlanLimitations();
        this.fetchUserKey();
      } catch (error) {
        console.error('Error loading accounts:', error);
      }
    }
  }

  /**
   * MÉTODO PRINCIPAL: Cargar TODAS las estrategias completas al cache
   * Este método carga tanto el overview como la configuration de cada estrategia
   * y las almacena en el cache para acceso rápido
   */
  private async loadAllStrategiesToCache(): Promise<void> {
    if (!this.user?.id) return;

    try {

      // Limpiar cache anterior
      this.strategyCacheService.clearCache();

      // 1. Obtener todas las estrategias (overviews)
      const allStrategies = await this.strategySvc.getUserStrategyViews(this.user.id);

      if (!allStrategies || allStrategies.length === 0) {
        this.userStrategies = [];
        this.activeStrategy = null;
        this.filteredStrategies = [];
        return;
      }

      // 2. Para cada estrategia, cargar su configuración completa
      const strategiesWithConfigs = await Promise.all(
        allStrategies.map(async (strategy) => {
          try {
            // Obtener la configuración completa
            const strategyData = await this.strategySvc.getStrategyView((strategy as
any).id);
            if (strategyData && strategyData.configuration) {
              // Almacenar en cache usando el servicio
              this.strategyCacheService.setStrategy(
                (strategy as any).id,
```

```
                strategyData.overview,
                strategyData.configuration
              );

              return {
                overview: strategyData.overview,
                configuration: strategyData.configuration
              };
            } else {
              return null;
            }
          } catch (error) {
            console.error(`'L Error loading strategy ${strategy.name}:`, error);
            return null;
          }
        })
      );

      // 3. Filtrar estrategias válidas y separar activa de inactivas
      const validStrategies = strategiesWithConfigs.filter(s => s !== null);

      this.activeStrategy = validStrategies.find(s => s.overview.status === true)?.overview
|| null;
      this.userStrategies = validStrategies.filter(s => s.overview.status !== true).map(s =>
s.overview);
      this.filteredStrategies = [...this.userStrategies];

      // 4. Cargar datos de las cards
      await this.loadStrategyCardsData();

      // 5. Actualizar strategy card si hay estrategia activa
      if (this.activeStrategy) {
        this.updateStrategyCardWithActiveStrategy();
      }

      // 6. Verificar limitaciones
      this.checkStrategyLimitations();

    } catch (error) {
      console.error(''L Error loading strategies to cache:', error);
      this.userStrategies = [];
      this.activeStrategy = null;
      this.filteredStrategies = [];
    }
  }

  /**
   * Inicializa los datos de reporte
   */
  private async initializeReportData(): Promise<void> {
    try {
      this.getActualBalance();
    } catch (error) {
      console.error('Error loading report data:', error);
    }
  }

  /**
   * Configurar todos los listeners necesarios
   */
  private setupListeners(): void {
    // Suscribirse a los datos del contexto
    this.subscribeToContextData();

    // Configurar listeners de reporte
    this.listenReportData();

    // Inicializar servicio global de actualización
    if (this.user?.id) {
      this.globalStrategyUpdater.updateAllStrategies(this.user.id);
    }
  }
```

```
  ngOnDestroy(): void {
    // Limpiar recursos si es necesario
  }

  fetchUserKey() {
    if (this.user?.email && this.accountsData.length > 0) {
      // Use the first account's credentials
      const firstAccount = this.accountsData[0];
      this.reportSvc
        .getUserKey(firstAccount.emailTradingAccount, firstAccount.brokerPassword, firstAccount.server)
        .subscribe({
          next: (key: string) => {
            this.store.dispatch(setUserKey({ userKey: key }));
          },
          error: (err) => {
            console.error('Error fetching user key:', err);
            this.store.dispatch(setUserKey({ userKey: '' }));
          },
        });
    } else {
      console.warn('No user email or accounts available for fetching user key');
      this.store.dispatch(setUserKey({ userKey: '' }));
    }
  }

  getActualBalance() {
    // Usar el método optimizado que carga desde cache primero
    this.loadConfigWithCachedBalance();
  }

  getUserData() {
    this.store.select(selectUser).subscribe({
      next: (user) => {
        this.user = user.user;
      },
      error: (err) => {
        console.error('Error fetching user data', err);
      },
    });
  }

  /**
   * MÉTODO SIMPLIFICADO: Cargar configuración usando el cache
   * Ya no hace peticiones a Firebase, solo lee del cache
   */
  async loadConfig(balance: number) {
    this.loading = true;

    try {
      // Si hay estrategia activa, usar su configuración del cache
      if (this.activeStrategy) {
        const cachedStrategy = this.strategyCacheService.getStrategy((this.activeStrategy as any).id);
        if (cachedStrategy) {
          // Actualizar balance en riskPerTrade
          const configWithBalance = {
            ...cachedStrategy.configuration,
            riskPerTrade: {
              ...cachedStrategy.configuration.riskPerTrade,
              balance: balance
            }
          };

          this.config = configWithBalance;
        } else {
          this.config = initialStrategyState;
        }
      } else {
        // No hay estrategia activa, usar configuración inicial
```

```
        this.config = {
          ...initialStrategyState,
          riskPerTrade: {
            ...initialStrategyState.riskPerTrade,
            balance: balance
          }
        };
      }

      await this.checkPlanLimitations();
    } catch (error) {
      console.error('Error loading config:', error);
      this.config = initialStrategyState;
    } finally {
      this.loading = false;
    }
  }

  /**
   * MÉTODO OPTIMIZADO: Cargar configuración con balance desde cache
   */
  async loadConfigWithCachedBalance() {
    this.loading = true;

    try {
      // Obtener balance desde cache
      let balance = 0;
      if (this.accountsData.length > 0) {
        const firstAccount = this.accountsData[0];
        balance = this.balanceCacheService.getBalance(firstAccount.accountID);
      }

      // Cargar configuración con balance del cache
      await this.loadConfig(balance);

      // Si necesita actualización, hacer petición en background
      if (this.accountsData.length > 0) {
        const firstAccount = this.accountsData[0];
        if (this.balanceCacheService.needsUpdate(firstAccount.accountID)) {
          this.updateBalanceInBackground(firstAccount);
        }
      }
    } catch (error) {
      console.error('Error loading config with cached balance:', error);
      this.config = initialStrategyState;
    } finally {
      this.loading = false;
    }
  }

  /**
   * Actualizar balance en background sin bloquear la UI
   */
  private updateBalanceInBackground(account: AccountData) {
    this.store.select(selectUserKey).pipe().subscribe({
      next: (userKey) => {
        if (userKey && userKey !== '') {
          this.reportSvc.getBalanceData(account.accountID as string, userKey,
account.accountNumber as number).subscribe({
            next: (balance) => {
              // Actualizar cache con nuevo balance
              this.balanceCacheService.setBalance(account.accountID, balance);
              // Recargar configuración con nuevo balance
              this.loadConfig(balance);
            },
            error: (err) => {
              console.error('Error fetching balance data in background:', err);
            },
          });
        }
      },
```

```
    });
  }

  /**
   * MÉTODO PÚBLICO: Invalidar cache y recargar estrategias
   * Se llama cuando hay cambios en las estrategias (crear, actualizar, eliminar)
   */
  public async invalidateCacheAndReload(): Promise<void> {
    await this.loadAllStrategiesToCache();
  }

  /**
   * MÉTODO PÚBLICO: Obtener estrategia del cache
   * Para que edit-strategy pueda acceder a los datos sin hacer peticiones
   */
  public getStrategyFromCache(strategyId: string): { overview: ConfigurationOverview;
configuration: StrategyState } | null {
    return this.strategyCacheService.getStrategy(strategyId);
  }

  /**
   * MÉTODO PÚBLICO: Verificar si el cache está cargado
   */
  public isCacheLoaded(): boolean {
    return this.strategyCacheService.isCacheLoaded();
  }

  openEditPopup() {
    this.router.navigate(['/edit-strategy']);
  }

  onSearchChange(event: Event) {
    const value = (event.target as HTMLInputElement).value;
    this.searchTerm = value;
    this.filterStrategies();
  }

  filterStrategies() {
    if (!this.searchTerm.trim()) {
      this.filteredStrategies = [...this.userStrategies];
    } else {
      this.filteredStrategies = this.userStrategies.filter(strategy =>
        strategy.name.toLowerCase().includes(this.searchTerm.toLowerCase())
      );
    }
  }

  // Obtener datos de card por ID de estrategia
  getCardDataByStrategyId(strategyId: string): StrategyCardData | undefined {
    return this.strategyCardsData.find(card => card.id === strategyId);
  }

  // Contar el total de estrategias del usuario (activas + inactivas) sin duplicar
  // IMPORTANTE: Solo cuenta estrategias NO eliminadas (deleted !== true)
  // Las estrategias eliminadas (deleted: true) NO se incluyen en este conteo
  private getTotalStrategiesCount(): number {
    const uniqueIds = new Set<string>();
    this.userStrategies.forEach(s => uniqueIds.add((s as any).id));
    if (this.activeStrategy) uniqueIds.add((this.activeStrategy as any).id);

    const total = uniqueIds.size;

    return total;
  }

  async onNewStrategy() {
    if (!this.user?.id || this.isCreatingStrategy) return;

    try {
      // Activar loading
      this.isCreatingStrategy = true;
```

```
586
587       // Contar el total de estrategias del usuario (activas + inactivas)
588       const totalStrategies = this.getTotalStrategiesCount();
589       const accessCheck = await
590 this.planLimitationsGuard.checkStrategyCreationWithModal(this.user.id, totalStrategies);
591       if (!accessCheck.canCreate) {
592         // Verificar si es el plan Pro con 8 estrategias (límite máximo)
593         const limitations = await
594 this.planLimitationsGuard.checkUserLimitations(this.user.id);
595         const isProPlanWithMaxStrategies =
596 limitations.planName.toLowerCase().includes('pro') &&
597                                            limitations.maxStrategies === 8 &&
598                                            totalStrategies >= 8;
599
600         if (isProPlanWithMaxStrategies) {
601           // Para plan Pro con 8 estrategias: desactivar botón
602           this.isAddStrategyDisabled = true;
603           return;
604         } else {
605           // Para otros planes: redirigir a la página de cuenta
606           this.router.navigate(['/account'], {
607             queryParams: { tab: 'plan' }
608           });
609           return;
610         }
611       }
612
613       // Verificar si es la primera estrategia del usuario
614       if (totalStrategies === 0) {
615         // Primera estrategia: mostrar modal de guía de estrategias
616         this.showStrategyGuide = true;
617       } else {
618         // Estrategias adicionales: crear automáticamente con nombre genérico
619         await this.createGenericStrategy();
620       }
621     } finally {
622       // Desactivar loading
623       this.isCreatingStrategy = false;
624     }
625   }
626
627   // Strategy Card Event Handlers
628   onStrategyEdit(strategyId: string) {
629     // Verificar si el usuario no ha marcado "don't show again"
630     const dontShowAgain = localStorage.getItem('strategy-guide-dont-show');
631
632     // Si no ha marcado "don't show again", mostrar el modal de guía
633     if (!dontShowAgain) {
634       this.showStrategyGuide = true;
635       return;
636     }
637
638     // Si ya marcó "don't show again", navegar directamente a edit-strategy
639     if (strategyId) {
640       this.router.navigate(['/edit-strategy'], { queryParams: { strategyId: strategyId } });
641     } else {
642       this.alertService.showWarning('No strategy found. Please create a strategy first.',
643 'No Strategy Found');
644     }
645   }
646
647   onStrategyFavorite(strategyId: string) {
648     // TODO: Implementar funcionalidad de favoritos en la base de datos
649     // Por ahora solo actualizar el estado local
650     const strategy = this.userStrategies.find(s => (s as any).id === strategyId);
651     if (strategy) {
652     }
653   }
654
655   onStrategyMoreOptions(strategyId: string) {
656     // TODO: Implementar menú de opciones (copiar, eliminar, etc.)
657     // Por ahora mostrar opciones básicas
658     const strategy = this.userStrategies.find(s => (s as any).id === strategyId);
```

```
    if (strategy) {
      const action = confirm(`Options for "${strategy.name}":\n\nOK - Copy strategy\nCancel - Delete strategy`);
      if (action) {
        this.copyStrategy(strategyId);
      } else {
        this.deleteStrategy(strategyId);
      }
    }
  }

  onStrategyCustomize(strategyId: string) {
    this.router.navigate(['/edit-strategy'], { queryParams: { strategyId: strategyId } });
  }

  // Plan detection and banner methods
  fetchUserAccounts() {
    if (this.user?.id) {
      this.authService.getUserAccounts(this.user.id).then(async (accounts) => {
        this.accountsData = accounts || [];
        await this.checkPlanLimitations();
        // After loading accounts, try to fetch user key
        this.fetchUserKey();
      });
    }
  }

  listenReportData() {
    // Listen to trade win percentage
    this.store.select(selectTradeWin).subscribe((tradeWin) => {
      this.tradeWin = tradeWin;
      this.updateStrategyCard();
    });

    // Listen to total trades
    this.store.select(selectTotalTrades).subscribe((totalTrades) => {
      this.totalTrades = totalTrades;
      this.updateStrategyCard();
    });

    // Listen to net PnL
    this.store.select(selectNetPnL).subscribe((netPnL) => {
      this.netPnL = netPnL;
      this.updateStrategyCard();
    });
  }

  updateStrategyCard() {
    // Solo actualizar si hay una estrategia activa
    if (this.activeStrategy) {
      this.updateStrategyCardWithActiveStrategy();
    }
  }

  countActiveRules(config: any): number {
    let count = 0;
    if (config.maxDailyTrades?.isActive) count++;
    if (config.riskReward?.isActive) count++;
    if (config.riskPerTrade?.isActive) count++;
    if (config.daysAllowed?.isActive) count++;
    if (config.hoursAllowed?.isActive) count++;
    if (config.assetsAllowed?.isActive) count++;
    return count;
  }

  getLastModifiedDate(): string {
    // For now, return current date. In the future, this could come from a timestamp in the config
    const now = new Date();
    const months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
    const month = months[now.getMonth()];
    const day = now.getDate();
```

```
    const year = now.getFullYear();
    return `${month} ${day}, ${year}`;
  }

  // Check if the current plan allows multiple strategies
  canCreateMultipleStrategies(): boolean {
    // This will be determined by the plan limitations guard
    return true; // Default to true, let the guard handle the actual validation
  }

  private getActiveStrategyCount(): number {
    if (!this.config) return 0;

    let count = 0;
    Object.values(this.config).forEach(rule => {
      if (rule && typeof rule === 'object' && 'isActive' in rule && rule.isActive) {
        count++;
      }
    });
    return count;
  }

  private async checkPlanLimitations() {
    if (!this.user?.id) {
      this.showPlanBanner = false;
      return;
    }

    try {
      // Get user's plan limitations from the guard
      const limitations = await this.planLimitationsGuard.checkUserLimitations(this.user.id);
      const totalStrategies = this.getTotalStrategiesCount();

      this.showPlanBanner = false;
      this.planBannerMessage = '';
      this.planBannerType = 'info';

      // If user needs subscription or is banned/cancelled
      if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
        // Only show banner if user has trading accounts (not first-time user with plan)
        if (this.accountsData.length > 0) {
          this.showPlanBanner = true;
          this.planBannerMessage = this.getBlockedMessage(limitations);
          this.planBannerType = 'warning';
        }
        return;
      }

      // Check if user has reached strategy limit
      if (totalStrategies >= limitations.maxStrategies) {
        this.showPlanBanner = true;
        this.planBannerMessage = `You've reached the strategy limit for your ${limitations.planName} plan. Move to a higher plan and keep growing your account.`;
        this.planBannerType = 'warning';
      }
    } catch (error) {
      console.error('Error checking plan limitations:', error);
      this.showPlanBanner = false;
    }
  }

  private getBlockedMessage(limitations: any): 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.';
  }

  onUpgradePlan() {
    this.router.navigate(['/account'], {
      queryParams: { tab: 'plan' }
    });
  }

  onCloseBanner() {
    this.showPlanBanner = false;
  }


  // Check strategy limitations and update button state
  async checkStrategyLimitations() {
    if (!this.user?.id) {
      this.isAddStrategyDisabled = true;
      return;
    }

    try {
      // Contar el total de estrategias del usuario (activas + inactivas)
      const totalStrategies = this.getTotalStrategiesCount();
      const accessCheck = await
this.planLimitationsGuard.checkStrategyCreationWithModal(this.user.id, totalStrategies);
      this.isAddStrategyDisabled = !accessCheck.canCreate;

      // El banner se actualiza automáticamente en checkPlanLimitations()
    } catch (error) {
      console.error('Error checking strategy limitations:', error);
      this.isAddStrategyDisabled = true;
    }
  }

  // ===== MÉTODOS PARA MÚLTIPLES ESTRATEGIAS =====


  // Cargar datos de las cards para todas las estrategias
  async loadStrategyCardsData() {
    this.strategyCardsData = [];

    for (const strategy of this.userStrategies) {
      try {
        const cardData = await this.getStrategyCardData(strategy);
        this.strategyCardsData.push(cardData);
      } catch (error) {
        // Agregar datos básicos en caso de error
        this.strategyCardsData.push({
          id: (strategy as any).id,
          name: strategy.name,
          status: strategy.status,
          lastModified: this.formatDate(strategy.updated_at.toDate()),
          rules: 0,
          days_active: strategy.days_active || 0,
          winRate: 0,
          isFavorite: false,
          created_at: strategy.created_at,
          updated_at: strategy.updated_at,
          userId: strategy.userId,
          configurationId: strategy.configurationId
        });
      }
    }
  }

  // Generar nombre único para estrategia
```

```
  private generateUniqueStrategyName(baseName: string): string {
    // Obtener todas las estrategias existentes (activas e inactivas)
    const allStrategies = [
      ...(this.activeStrategy ? [this.activeStrategy] : []),
      ...this.userStrategies
    ];

    // Extraer solo los nombres
    const existingNames = allStrategies.map(strategy => strategy.name);

    // Si el nombre base no existe, usarlo tal como está
    if (!existingNames.includes(baseName)) {
      return baseName;
    }

    // Si el nombre base termina con "copy", agregar número secuencial
    if (baseName.toLowerCase().endsWith('copy')) {
      let counter = 1;
      let newName = `${baseName} ${counter}`;

      while (existingNames.includes(newName)) {
        counter++;
        newName = `${baseName} ${counter}`;
      }

      return newName;
    }

    // Si el nombre base no termina con "copy", agregar "copy" primero
    let copyName = `${baseName} copy`;

    if (!existingNames.includes(copyName)) {
      return copyName;
    }

    // Si "copy" ya existe, agregar número secuencial
    let counter = 1;
    let newName = `${baseName} copy ${counter}`;

    while (existingNames.includes(newName)) {
      counter++;
      newName = `${baseName} copy ${counter}`;
    }

    return newName;
  }



  // Crear estrategia genérica automáticamente (para estrategias adicionales)
  async createGenericStrategy() {
    if (!this.user?.id) return;

    try {
      // 1. Primero recargar las strategies para tener el estado actualizado
      await this.invalidateCacheAndReload();

      // 2. Generar nombre único para la estrategia genérica
      const genericName = this.generateUniqueStrategyName('Strategy');

      // 3. Verificar si ya hay una estrategia activa
      const hasActiveStrategy = this.activeStrategy !== null;

      // 4. Verificar si es la primera estrategia del usuario
      // NOTA: getTotalStrategiesCount() solo cuenta estrategias NO eliminadas (deleted !==
true) // Las estrategias eliminadas no cuentan para determinar si es la primera
      const totalStrategies = this.getTotalStrategiesCount();
      const isFirstStrategy = totalStrategies === 0;

      // 5. Crear configuración vacía con reglas por defecto (todas inactivas)
```

```
      const emptyStrategyConfig: StrategyState = {
        maxDailyTrades: {
          isActive: false,
          maxDailyTrades: 0,
          type: 'MAX DAILY TRADES' as any,
        },
        riskReward: {
          isActive: false,
          riskRewardRatio: '1:2',
          type: 'RISK REWARD RATIO' as any,
        },
        riskPerTrade: {
          isActive: false,
          review_type: 'MAX',
          number_type: 'PERCENTAGE',
          percentage_type: 'NULL',
          risk_ammount: 0,
          type: 'MAX RISK PER TRADE' as any,
          balance: 0,
          actualBalance: 0,
        },
        daysAllowed: {
          isActive: false,
          type: 'DAYS ALLOWED' as any,
          tradingDays: [],
        },
        hoursAllowed: {
          isActive: false,
          tradingOpenTime: '',
          tradingCloseTime: '',
          timezone: '',
          type: 'TRADING HOURS' as any,
        },
        assetsAllowed: {
          isActive: false,
          type: 'ASSETS ALLOWED' as any,
          assetsAllowed: [],
        },
      };

      // 6. Crear la nueva estrategia genérica
      // La primera estrategia siempre es activa, las adicionales siempre inactivas
      const strategyId = await this.strategySvc.createStrategyView(
        this.user.id,
        genericName,
        emptyStrategyConfig,
        isFirstStrategy ? true : false // Primera estrategia activa, el resto inactivas
      );

      // 6.1. Si NO es la primera estrategia, agregar dateInactive inmediatamente
      // Esto evita problemas en los reports al tener estrategias con dateActive sin
dateInactive
      if (!isFirstStrategy) {
        const inactiveTime = new Date(Date.now() + 2000); // +2 segundos desde ahora

        // Solo agregar dateInactive para cerrar el ciclo de activación/desactivación
        await this.strategySvc.updateStrategyDates(
          this.user.id,
          strategyId,
          undefined, // No agregar dateActive (ya existe uno)
          inactiveTime // Agregar dateInactive en 2 segundos
        );
      }

      // 7. Actualizar el estado del plan en tiempo real después de crear
      await this.checkPlanLimitations();

      // 8. Redirigir directamente a edit-strategy con la nueva estrategia
      this.router.navigate(['/edit-strategy'], { queryParams: { strategyId: strategyId } });

    } catch (error) {
```

```
      console.error('Error creating generic strategy:', error);
      this.alertService.showError('Error creating strategy. Please try again.', 'Strategy Creation Error');
    }
  }

  // Activar una estrategia
  async activateStrategy(strategyId: string) {
    if (!this.user?.id) return;

    try {
      // Mostrar loading completo durante la activación
      this.isProcessingStrategy = true;

      const currentTimestamp = new Date();

      // Verificar si hay una estrategia activa actualmente
      if (this.activeStrategy && (this.activeStrategy as any).id !== strategyId) {
        // Hay una estrategia activa diferente, desactivarla primero
        await this.strategySvc.updateStrategyDates(
          this.user.id,
          (this.activeStrategy as any).id,
          undefined, // No agregar a dateActive
          currentTimestamp // Agregar timestamp actual a dateInactive
        );
      }

      // Activar la nueva estrategia
      await this.strategySvc.updateStrategyDates(
        this.user.id,
        strategyId,
        currentTimestamp, // Agregar timestamp actual a dateActive
        undefined // No agregar a dateInactive
      );

      // Invalidar cache y recargar estrategias
      await this.invalidateCacheAndReload();

      // Recargar configuración con la nueva estrategia activa
      // Obtener balance desde la configuración actual
      const currentConfig = this.config;
      const balance = currentConfig?.riskPerTrade?.balance || 0;
      this.loadConfig(balance);
    } catch (error) {
      console.error('Error activating strategy:', error);
      this.alertService.showError('Error activating strategy. Please try again.', 'Strategy Activation Error');
    } finally {
      // Ocultar loading al finalizar
      this.isProcessingStrategy = false;
    }
  }

  // Eliminar estrategia
  deleteStrategy(strategyId: string) {
    // Guardar el ID de la estrategia a eliminar y mostrar el popup de confirmación
    this.strategyToDeleteId = strategyId;
    this.showDeleteConfirmPopup = true;
  }

  // Confirmar eliminación de estrategia (marcar como deleted)
  confirmDeleteStrategy = async () => {
    this.showDeleteConfirmPopup = false;

    try {
      // Mostrar loading completo durante la eliminación
      this.isProcessingStrategy = true;

      // Marcar la estrategia como deleted en lugar de borrarla
      await this.strategySvc.markStrategyAsDeleted(this.strategyToDeleteId);

      // Invalidar cache y recargar estrategias
```

```
      await this.invalidateCacheAndReload();

      // Actualizar el estado del plan en tiempo real después de eliminar
      await this.checkPlanLimitations();

      // Verificar si se debe reactivar el botón (para plan Pro que ya no está en el límite máximo)
      if (this.user?.id) {
        const limitations = await this.planLimitationsGuard.checkUserLimitations(this.user.id);
        const currentTotalStrategies = this.userStrategies.length + (this.activeStrategy ? 1 : 0);
        const isProPlan = limitations.planName.toLowerCase().includes('pro') && limitations.maxStrategies === 8;
        if (isProPlan && currentTotalStrategies < 8) {
          // Reactivar el botón si ya no está en el límite máximo
          this.isAddStrategyDisabled = false;
        }
      }

      // Si se eliminó la estrategia activa, cargar la primera disponible o estado inicial
      if (!this.activeStrategy) {
        if (this.userStrategies.length > 0) {
          await this.activateStrategy((this.userStrategies[0] as any).id);
        } else {
          this.config = initialStrategyState;
        }
      }
    } catch (error) {
      this.alertService.showError('Error marking strategy as deleted. Please try again.', 'Strategy Deletion Error');
    } finally {
      // Ocultar loading al finalizar
      this.isProcessingStrategy = false;
      this.strategyToDeleteId = '';
    }
  };

  // Cancelar eliminación de estrategia
  cancelDeleteStrategy = () => {
    this.showDeleteConfirmPopup = false;
    this.strategyToDeleteId = '';
  };

  // Actualizar nombre de estrategia
  async updateStrategyName(strategyId: string, newName: string) {
    if (!newName.trim()) {
      this.alertService.showWarning('Please enter a valid strategy name', 'Invalid Strategy Name');
      return;
    }

    try {
      await this.strategySvc.updateStrategyView(strategyId, { name: newName.trim() });

      // Invalidar cache y recargar estrategias
      await this.invalidateCacheAndReload();
    } catch (error) {
      this.alertService.showError('Error updating strategy name. Please try again.', 'Strategy Name Update Error');
    }
  }

  // Actualizar strategy card con la estrategia activa
  async updateStrategyCardWithActiveStrategy() {
    if (!this.activeStrategy || !this.user?.id) return;

    try {
      // Obtener la estrategia completa (configurations + configuration-overview)
      const strategyData = await this.strategySvc.getStrategyView((this.activeStrategy as any).id);
      if (!strategyData) {
        console.error('No strategy data found for active strategy');
        return;
      }
```

```
      // Calcular win rate (por ahora 0, necesitamos implementar la lógica)
      const winRate = 0; // TODO: Implementar cálculo de win rate

      // Contar reglas activas de la configuración
      const activeRules = strategyData.configuration ?
this.countActiveRules(strategyData.configuration) : 0;
      // Formatear fecha de actualización
      const lastModified = this.formatDate(strategyData.overview.updated_at.toDate());

      this.strategyCard = {
        id: (this.activeStrategy as any).id,
        name: strategyData.overview.name, // Nombre de la estrategia desde overview
        status: strategyData.overview.status, // Estado desde overview
        lastModified: lastModified, // updated_at formateado desde overview
        rules: activeRules, // Reglas activas de configuration
        days_active: strategyData.overview.days_active || 0, // days_active desde overview
        winRate: winRate, // Win rate (por implementar)
        isFavorite: this.strategyCard.isFavorite,
        created_at: strategyData.overview.created_at,
        updated_at: strategyData.overview.updated_at,
        userId: strategyData.overview.userId,
        configurationId: strategyData.overview.configurationId
      };
    } catch (error) {
      // Error silencioso
    }
  }

  // Formatear fecha
  formatDate(date: Date): string {
    const months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
'Nov', 'Dec'];
    const month = months[date.getMonth()];
    const day = date.getDate();
    const year = date.getFullYear();
    return `${month} ${day}, ${year}`;
  }




  // Obtener ID de la estrategia para el track
  getStrategyId(strategy: ConfigurationOverview): string {
    return (strategy as any).id;
  }

  // Convertir ConfigurationOverview a StrategyCardData
  async getStrategyCardData(strategy: ConfigurationOverview): Promise<StrategyCardData> {
    try {
      // Obtener la estrategia completa (configurations + configuration-overview)
      const strategyData = await this.strategySvc.getStrategyView((strategy as any).id);

      if (!strategyData) {
        // Retornar datos básicos en caso de error
        return {
          id: (strategy as any).id,
          name: strategy.name,
          status: strategy.status,
          lastModified: this.formatDate(strategy.updated_at.toDate()),
          rules: 0,
          days_active: strategy.days_active || 0,
          winRate: 0,
          isFavorite: false,
          created_at: strategy.created_at,
          updated_at: strategy.updated_at,
          userId: strategy.userId,
          configurationId: strategy.configurationId
        };
      }

      // Calcular win rate (por ahora 0, necesitamos implementar la lógica)
```

```
      const winRate = 0; // TODO: Implementar cálculo de win rate

      // Contar reglas activas de la configuración
      const activeRules = strategyData.configuration ?
this.countActiveRules(strategyData.configuration) : 0;
      // Formatear fecha de actualización
      const lastModified = this.formatDate(strategyData.overview.updated_at.toDate());

      const cardData: StrategyCardData = {
        id: (strategy as any).id,
        name: strategyData.overview.name, // Nombre desde overview
        status: strategyData.overview.status, // Estado desde overview
        lastModified: lastModified, // updated_at desde overview
        rules: activeRules, // Reglas activas de configuration
        days_active: strategyData.overview.days_active || 0, // days_active desde overview
        winRate: winRate, // Win rate (por implementar)
        isFavorite: false,
        created_at: strategyData.overview.created_at,
        updated_at: strategyData.overview.updated_at,
        userId: strategyData.overview.userId,
        configurationId: strategyData.overview.configurationId
      };

      return cardData;
    } catch (error) {
      // Retornar datos básicos en caso de error
      return {
        id: (strategy as any).id,
        name: strategy.name,
        status: strategy.status,
        lastModified: this.formatDate(strategy.updated_at.toDate()),
        rules: 0,
        days_active: strategy.days_active || 0,
        winRate: 0,
        isFavorite: false,
        created_at: strategy.created_at,
        updated_at: strategy.updated_at,
        userId: strategy.userId,
        configurationId: strategy.configurationId
      };
    }
  }

  // Copiar estrategia
  async copyStrategy(strategyId: string) {
    if (!this.user?.id) return;

    try {
      // Mostrar loading completo durante la copia
      this.isProcessingStrategy = true;

      // Verificar límites del plan antes de duplicar
      const totalStrategies = this.getTotalStrategiesCount();
      const accessCheck = await
this.planLimitationsGuard.checkStrategyCreationWithModal(this.user.id, totalStrategies);
      if (!accessCheck.canCreate) {
        // Verificar si es el plan Pro con 8 estrategias (límite máximo)
        const limitations = await
this.planLimitationsGuard.checkUserLimitations(this.user.id);
        const isProPlanWithMaxStrategies =
limitations.planName.toLowerCase().includes('pro') &&
                                           limitations.maxStrategies === 8 &&
                                           totalStrategies >= 8;

        if (isProPlanWithMaxStrategies) {
          // Para plan Pro con 8 estrategias: desactivar botón y mostrar mensaje
          this.isAddStrategyDisabled = true;
          this.alertService.showWarning('You have reached the maximum number of strategies
(8) for your Pro plan.', 'Strategy Limit Reached');
          return;
        } else {
          // Para otros planes: redirigir a la página de cuenta
          this.router.navigate(['/account'], {
```

```
            queryParams: { tab: 'plan' }
          });
          return;
        }
      }

      // Buscar en estrategias inactivas primero
      let strategy = this.userStrategies.find(s => (s as any).id === strategyId);

      // Si no se encuentra en inactivas, buscar en la estrategia activa
      if (!strategy && this.activeStrategy && (this.activeStrategy as any).id === strategyId) {
        strategy = this.activeStrategy;
      }

      if (!strategy) {
        console.error('Strategy not found');
        return;
      }

      // Determinar el nombre de la copia usando la lógica de nombres únicos
      const isActiveStrategy = this.activeStrategy && (this.activeStrategy as any).id === strategyId;
      const baseName = strategy.name;
      const newName = this.generateUniqueStrategyName(baseName);

      // Obtener la estrategia completa (configuration-overview + configurations)
      const strategyData = await this.strategySvc.getStrategyView((strategy as any).id);
      if (!strategyData || !strategyData.configuration) {
        console.error('Strategy configuration not found');
        return;
      }

      // Crear configuración con los campos requeridos para la copia
      const strategyConfig: StrategyState = {
        ...strategyData.configuration
      };

      // Si es la estrategia activa, crear la copia como inactiva
      const newStrategyId = await this.strategySvc.createStrategyView(
        this.user.id,
        newName,
        strategyConfig,
        isActiveStrategy ? false : undefined // false = inactiva, undefined = mantener estado original
      );

      // Si se está copiando una estrategia activa (isActiveStrategy = true),
      // agregar dateInactive inmediatamente para cerrar el ciclo de activación
      if (isActiveStrategy) {
        const inactiveTime = new Date(Date.now() + 2000); // +2 segundos desde ahora

        // Solo agregar dateInactive para cerrar el ciclo de activación/desactivación
        await this.strategySvc.updateStrategyDates(
          this.user.id,
          newStrategyId,
          undefined, // No agregar dateActive (ya existe uno)
          inactiveTime // Agregar dateInactive en 2 segundos
        );
      }

      // Invalidar cache y recargar estrategias
      await this.invalidateCacheAndReload();

      // Actualizar el estado del plan en tiempo real después de copiar
      await this.checkPlanLimitations();

      // Verificar si se debe reactivar el botón (para plan Pro que ya no está en el límite máximo)
      if (this.user?.id) {
        const limitations = await this.planLimitationsGuard.checkUserLimitations(this.user.id);
        const currentTotalStrategies = this.userStrategies.length + (this.activeStrategy ? 1 : 0);
        const isProPlan = limitations.planName.toLowerCase().includes('pro') && limitations.maxStrategies === 8;
```

```
        if (isProPlan && currentTotalStrategies < 8) {
          // Reactivar el botón si ya no está en el límite máximo
          this.isAddStrategyDisabled = false;
        }
      }

    } catch (error) {
      this.alertService.showError('Error copying strategy. Please try again.', 'Strategy
Copy Error');
    } finally {
      // Ocultar loading al finalizar
      this.isProcessingStrategy = false;
    }
  }

  // Navegar a trading accounts
  navigateToTradingAccounts() {
    const accountsCount = this.accountsData?.length || 0;
    if (accountsCount >= 8) {
      return; // botón ya estará deshabilitado; no hacer nada
    }
    // Redirigir a Plan Management en Account
    this.router.navigate(['/account'], { queryParams: { tab: 'plan' } });
  }

  // Strategy guide modal methods
  private checkShowStrategyGuide(): void {
    const dontShowAgain = localStorage.getItem('strategy-guide-dont-show');
    if (!dontShowAgain) {
      this.showStrategyGuide = true;
    }
  }

  onCloseStrategyGuide(): void {
    this.showStrategyGuide = false;
  }

  onDontShowStrategyGuideAgain(): void {
    localStorage.setItem('strategy-guide-dont-show', 'true');
    this.showStrategyGuide = false;
  }

  async onEditStrategyFromGuide(): Promise<void> {
    // Cerrar el modal de guía
    this.showStrategyGuide = false;

    // Activar loading
    this.isCreatingStrategy = true;

    try {
      // Crear automáticamente la primera estrategia genérica
      await this.createGenericStrategy();
    } finally {
      // Desactivar loading
      this.isCreatingStrategy = false;
    }
  }
}
```

Ø=ÜÁ features\strategy\components\assets-allowed

Ø=ÜÄ features\strategy\components\assets-allowed\assets-allowed.component.ts

---

```ts
import { Component, HostListener, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../service/strategy.service';
import {
  assetsAllowed,
  daysAllowed,
  selectMaxDailyTrades,
} from '../../store/strategy.selectors';
import {
  AssetsAllowedConfig,
  Days,
  DaysAllowedConfig,
  MaxDailyTradesConfig,
  RuleType,
} from '../../models/strategy.model';
import {
  setAssetsAllowedConfig,
  setDaysAllowedConfig,
  setMaxDailyTradesConfig,
} from '../../store/strategy.actions';
import { CommonModule } from '@angular/common';

/**
 * Component for configuring the assets allowed for trading rule.
 *
 * This component allows users to select which trading instruments/assets are
 * permitted for trading. It includes a searchable dropdown for selecting
 * instruments and displays selected assets as removable chips.
 *
 * Features:
 * - Toggle rule active/inactive
 * - Searchable dropdown for instrument selection
 * - Display selected assets as chips
 * - Remove asset functionality
 * - Syncs with NgRx store
 *
 * Relations:
 * - Store (NgRx): Reads and updates assetsAllowed configuration
 * - SettingsService: Strategy service (injected but not directly used)
 *
 * @component
 * @selector app-assets-allowed
 * @standalone true
 */
@Component({
  selector: 'app-assets-allowed',
  templateUrl: './assets-allowed.component.html',
  styleUrls: ['./assets-allowed.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class AssetsAllowedComponent implements OnInit {
  config: AssetsAllowedConfig = {
    isActive: false,
    type: RuleType.ASSETS_ALLOWED,
    assetsAllowed: ['XMRUSD', 'BTCUSD'],
  };

  symbols: string[] = [];
  availableSymbolsOptions: string[] = [];
  selectedInstrument: string | undefined = undefined;
  searchTerm: string = '';

  dropdownOpen = false;

  constructor(private store: Store, private settingsService: SettingsService) {}

  closeDropdown() {
    this.dropdownOpen = false;
  }
```

```
  toggleDropdown() {
    if (this.config.isActive) {
      this.dropdownOpen = !this.dropdownOpen;
    } else {
      this.dropdownOpen = false;
    }
  }

  onSearchInput(event: Event) {
    this.searchTerm = (event.target as HTMLInputElement).value;
    this.dropdownOpen = true;
  }

  onSearchFocus() {
    if (this.config.isActive) {
      this.dropdownOpen = true;
    }
  }

  onSearchBlur() {
    // Delay para permitir click en dropdown
    setTimeout(() => {
      this.dropdownOpen = false;
    }, 200);
  }

  selectInstrument(intrument: string) {
    this.selectedInstrument = intrument;
    this.addSymbol(intrument);
    this.dropdownOpen = false;
    this.selectedInstrument = undefined;
    this.searchTerm = ''; // Limpiar búsqueda
  }

  getFilteredSymbols(): string[] {
    if (!this.searchTerm) {
      return this.availableSymbolsOptions;
    }
    return this.availableSymbolsOptions.filter(symbol =>
      symbol.toLowerCase().includes(this.searchTerm.toLowerCase())
    );
  }

  ngOnInit(): void {
    this.listenRuleConfiguration();
  }

  addSymbol(symbol: string) {
    if (symbol && !this.symbols.includes(symbol)) {
      this.symbols = [...this.symbols, symbol];
    }
    this.updateConfig({
      ...this.config,
      assetsAllowed: this.symbols,
    });
  }

  removeSymbol(symbol: string) {
    if (this.config.isActive) {
      this.symbols = this.symbols.filter((s) => s !== symbol);
      this.updateConfig({
        ...this.config,
        assetsAllowed: this.symbols,
      });
    }
  }

  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
```

```
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Reiniciar assets cuando se desactiva
      assetsAllowed: isActive ? this.config.assetsAllowed : [],
    };

    // Reiniciar símbolos seleccionados
    if (!isActive) {
      this.symbols = [];
    }

    this.updateConfig(newConfig);
  }

  listenRuleConfiguration() {
    this.store
      .select(assetsAllowed)
      .pipe()
      .subscribe((config) => {
        this.config = { ...config };
        this.symbols = this.config.assetsAllowed;
      });
  }

  private updateConfig(config: AssetsAllowedConfig) {
    this.store.dispatch(setAssetsAllowedConfig({ config }));
  }
}

```

Ø=ÜÁ features\strategy\components\days-allowed

Ø=ÜÄ features\strategy\components\days-allowed\days-allowed.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../service/strategy.service';
import {
  daysAllowed,
  selectMaxDailyTrades,
} from '../../store/strategy.selectors';
import {
  Days,
  DaysAllowedConfig,
  MaxDailyTradesConfig,
  RuleType,
} from '../../models/strategy.model';
import {
  setDaysAllowedConfig,
  setMaxDailyTradesConfig,
} from '../../store/strategy.actions';
import { CommonModule } from '@angular/common';

/**
 * Component for configuring the days allowed for trading rule.
 *
 * This component allows users to select which days of the week trading is
 * permitted. It displays buttons for each day (Monday through Sunday) that
 * can be toggled on/off.
 *
 * Features:
 * - Toggle rule active/inactive
```

```
 * - Day buttons for each day of the week
 * - Visual feedback for selected days
 * - Syncs with NgRx store
 *
 * Relations:
 * - Store (NgRx): Reads and updates daysAllowed configuration
 * - SettingsService: Strategy service (injected but not directly used)
 *
 * @component
 * @selector app-days-allowed
 * @standalone true
 */
@Component({
  selector: 'app-days-allowed',
  templateUrl: './days-allowed.component.html',
  styleUrls: ['./days-allowed.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class DaysAllowedComponent implements OnInit {
  config: DaysAllowedConfig = {
    isActive: false,
    type: RuleType.DAYS_ALLOWED,
    tradingDays: [],
  };

  daysButtons = [
    { day: Days.MONDAY, isActive: false },
    { day: Days.TUESDAY, isActive: false },
    { day: Days.WEDNESDAY, isActive: false },
    { day: Days.THURSDAY, isActive: false },
    { day: Days.FRIDAY, isActive: false },
    { day: Days.SATURDAY, isActive: false },
    { day: Days.SUNDAY, isActive: false },
  ];

  constructor(private store: Store, private settingsService: SettingsService) {}

  ngOnInit(): void {
    this.listenRuleConfiguration();
  }
  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Reiniciar días cuando se desactiva
      tradingDays: isActive ? this.config.tradingDays : [],
    };

    // Reiniciar botones de días
    if (!isActive) {
      this.daysButtons.forEach(day => {
        day.isActive = false;
      });
    }

    this.updateConfig(newConfig);
  }

  onChangeValue(day: { day: Days; isActive: boolean }) {
    if (this.config.isActive) {
      this.daysButtons.forEach((d) => {
        if (d.day === day.day) {
          d.isActive = !d.isActive;
        }
      });

      const newConfig: DaysAllowedConfig = {
        ...this.config,
```

```
        tradingDays: this.transformDaysActive(),
      };
      this.updateConfig(newConfig);
    }
  }

  transformDaysActive(): string[] {
    let daysArr: string[] = [];

    this.daysButtons.forEach((d) => {
      if (d.isActive) {
        daysArr.push(d.day);
      }
    });

    return daysArr;
  }

  listenRuleConfiguration() {
    this.store
      .select(daysAllowed)
      .pipe()
      .subscribe((config) => {
        this.config = config;
        this.daysButtons.forEach((dayOption) => {
          const findedDay = config.tradingDays.find((d) => d === dayOption.day);
          if (findedDay) {
            dayOption.isActive = true;
          }
        });
      });
  }

  private updateConfig(config: DaysAllowedConfig) {
    this.store.dispatch(setDaysAllowedConfig({ config }));
  }
}

```

Ø=ÜÁ features\strategy\components\hours-allowed

Ø=ÜÄ features\strategy\components\hours-allowed\hours-allowed.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../service/strategy.service';
import {
  hoursAllowed,
  selectMaxDailyTrades,
} from '../../store/strategy.selectors';
import {
  HoursAllowedConfig,
  MaxDailyTradesConfig,
  RuleType,
} from '../../models/strategy.model';
import {
  setHoursAllowedConfig,
  setMaxDailyTradesConfig,
} from '../../store/strategy.actions';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { NgxMaterialTimepickerModule } from 'ngx-material-timepicker';
import * as moment from 'moment-timezone';
```

```
import { AlertService } from '../../../../shared/services/alert.service';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

/**
 * Component for configuring the trading hours allowed rule.
 *
 * This component allows users to set the time window during which trading is
 * permitted. It includes time pickers for opening and closing times, and a
 * timezone selector with all available timezones.
 *
 * Features:
 * - Toggle rule active/inactive
 * - Time pickers for opening and closing times
 * - Timezone selector with GMT offsets
 * - Validation: opening time must be before closing time
 * - Minimum 30-minute difference between times
 *
 * Relations:
 * - Store (NgRx): Reads and updates hoursAllowed configuration
 * - NgxMaterialTimepickerModule: Time picker UI
 * - AlertService: Shows validation warnings
 *
 * @component
 * @selector app-hours-allowed
 * @standalone true
 */
@Component({
  selector: 'app-hours-allowed',
  templateUrl: './hours-allowed.component.html',
  styleUrls: ['./hours-allowed.component.scss'],
  imports: [CommonModule, FormsModule, NgxMaterialTimepickerModule],
  standalone: true,
})
export class HoursAllowedComponent implements OnInit {
  config: HoursAllowedConfig = {
    isActive: false,
    tradingOpenTime: '09:30',
    tradingCloseTime: '17:00',
    timezone: 'Zulu',
    type: RuleType.TRADING_HOURS,
  };

  timezones = Array.from(
    new Map(
      moment.tz.names().map((tz) => {
        const offset = moment.tz(tz).utcOffset();
        const offsetSign = offset >= 0 ? '+' : '-';
        const absOffset = Math.abs(offset);
        const hours = Math.floor(absOffset / 60);
        const mins = absOffset % 60;

        const formattedOffset = `(GMT${offsetSign}${hours
          .toString()
          .padStart(2, '0')}:${mins.toString().padStart(2, '0')})`;

        const abbreviation = moment.tz(tz).zoneAbbr();
        const key = `${abbreviation} ${formattedOffset}`;

        return [key, { value: tz, label: key, offsetMinutes: offset }] as [
          string,
          { value: string; label: string; offsetMinutes: number }
        ];
      })
    ).values()
  )
    .sort((a, b) => a.offsetMinutes - b.offsetMinutes)
    .map(({ value, label }) => ({ value, label }));

  constructor(private store: Store, private settingsService: SettingsService, private
alertService: AlertService) {}
```

```
  ngOnInit(): void {
    this.listenRuleConfiguration();
  }

  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Reiniciar valores cuando se desactiva
      tradingOpenTime: isActive ? this.config.tradingOpenTime : '09:30',
      tradingCloseTime: isActive ? this.config.tradingCloseTime : '17:00',
      timezone: isActive ? this.config.timezone : 'UTC',
    };
    this.updateConfig(newConfig);
  }
  onTimezoneChange(newTz: string) {
    // Validar que la timezone sea válida
    if (this.isValidTimezone(newTz)) {
      const newConfig = { ...this.config, timezone: newTz };
      this.updateConfig(newConfig);
    } else {
      console.warn('Invalid timezone selected:', newTz);
    }
  }

  isValidTimezone(timezone: string): boolean {
    return this.timezones.some(tz => tz.value === timezone);
  }

  onTimeChange(field: 'tradingOpenTime' | 'tradingCloseTime', value: string) {
    const tempConfig = { ...this.config, [field]: value };
    const openMinutes = this.toMinutes(tempConfig.tradingOpenTime);
    const closeMinutes = this.toMinutes(tempConfig.tradingCloseTime);
    if (openMinutes >= closeMinutes) {
      this.alertService.showWarning('Opening time must be earlier than closing time.',
'Invalid Time Range');
      return;
    }
    if (closeMinutes - openMinutes < 30) {
      this.alertService.showWarning('There must be at least a 30-minute difference between
opening and closing times.', 'Minimum Time Difference');
      return;
    }
    this.updateConfig(tempConfig);
  }

  listenRuleConfiguration() {
    this.store
      .select(hoursAllowed)
      .pipe()
      .subscribe((config) => {
        this.config = { ...config };
      });
  }

  private updateConfig(config: HoursAllowedConfig) {
    this.store.dispatch(setHoursAllowedConfig({ config }));
  }

  private toMinutes(time: string): number {
    const is12hFormat =
      time.toUpperCase().includes('AM') || time.toUpperCase().includes('PM');

    let hours: number;
    let minutes: number;

    if (is12hFormat) {
      const [timePart, period] = time.split(' ');
      [hours, minutes] = timePart.split(':').map(Number);
      if (period.toUpperCase() === 'PM' && hours !== 12) {
        hours += 12;
```

```
      }
      if (period.toUpperCase() === 'AM' && hours === 12) {
        hours = 0;
      }
    } else {
      [hours, minutes] = time.split(':').map(Number);
    }

    return hours * 60 + minutes;
  }
}

```

Ø=ÜÁ features\strategy\components\max-daily-trades

Ø=ÜÄ features\strategy\components\max-daily-trades\max-daily-trades.component.ts

```typescript
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../service/strategy.service';
import { selectMaxDailyTrades } from '../../store/strategy.selectors';
import { MaxDailyTradesConfig, RuleType } from '../../models/strategy.model';
import { setMaxDailyTradesConfig } from '../../store/strategy.actions';
import { CommonModule } from '@angular/common';

/**
 * Component for configuring the maximum daily trades rule.
 *
 * This component allows users to set the maximum number of trades that can be
 * executed per day. It includes increment/decrement buttons and ensures the
 * minimum value is 1.
 *
 * Features:
 * - Toggle rule active/inactive
 * - Number input with validation (minimum 1)
 * - Increment/decrement buttons
 * - Syncs with NgRx store
 *
 * Relations:
 * - Store (NgRx): Reads and updates maxDailyTrades configuration
 * - SettingsService: Strategy service (injected but not directly used)
 *
 * @component
 * @selector app-max-daily-trades
 * @standalone true
 */
@Component({
  selector: 'app-max-daily-trades',
  templateUrl: './max-daily-trades.component.html',
  styleUrls: ['./max-daily-trades.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class MaxDailyTradesComponent implements OnInit {
  config: MaxDailyTradesConfig = {
    isActive: false,
    maxDailyTrades: 1,
    type: RuleType.MAX_DAILY_TRADES,
  };

  constructor(private store: Store, private settingsService: SettingsService) {}

  ngOnInit(): void {
```

```
    this.listenRuleConfiguration();
  }
  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Mantener el valor actual, no resetear
    };
    this.updateConfig(newConfig);
  }

  onChangeValue(event: Event) {
    const numValue = Number((event.target as HTMLInputElement).value);
    const newConfig: MaxDailyTradesConfig = {
      ...this.config,
      maxDailyTrades: numValue < 1 ? 1 : numValue,
    };
    this.updateConfig(newConfig);
  }

  // Métodos para spinner (solo incrementar/decrementar)
  incrementValue() {
    if (this.config.isActive) {
      const newConfig: MaxDailyTradesConfig = {
        ...this.config,
        maxDailyTrades: this.config.maxDailyTrades + 1,
      };
      this.updateConfig(newConfig);
    }
  }

  decrementValue() {
    if (this.config.isActive && this.config.maxDailyTrades > 1) {
      const newConfig: MaxDailyTradesConfig = {
        ...this.config,
        maxDailyTrades: this.config.maxDailyTrades - 1,
      };
      this.updateConfig(newConfig);
    }
  }

  listenRuleConfiguration() {
    this.store
      .select(selectMaxDailyTrades)
      .pipe()
      .subscribe((config) => {
        this.config = config;
      });
  }

  private updateConfig(config: MaxDailyTradesConfig) {
    this.store.dispatch(setMaxDailyTradesConfig({ config }));
  }
}

```

Ø=ÜÁ features\strategy\components\risk-per-trade

Ø=ÜÄ features\strategy\components\risk-per-trade\risk-per-trade.component.ts

```
import { CommonModule } from '@angular/common';
import { Component, HostListener, OnInit } from '@angular/core';
```

```typescript
import {
  MaxDailyTradesConfig,
  RiskPerTradeConfig,
  RiskRewardConfig,
  RuleType,
} from '../../models/strategy.model';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../service/strategy.service';
import { riskPerTrade, riskReward } from '../../store/strategy.selectors';
import {
  setRiskPerTradeConfig,
  setRiskRewardConfig,
} from '../../store/strategy.actions';
import { currencies } from './models/risk-per-trade.model';
import { AppContextService } from '../../../../shared/context';
import { AuthService } from '../../../auth/service/authService';
import { ReportService } from '../../../report/service/report.service';

/**
 * Component for configuring the maximum risk per trade rule.
 *
 * This component allows users to set the maximum amount of risk that can be
 * taken per trade. It supports multiple configuration options:
 * - Review type: MAX (maximum allowed) or FIXED (fixed amount)
 * - Number type: PERCENTAGE or MONEY
 * - Percentage type: INITIAL_B (initial balance), ACTUAL_B (actual balance), or NULL
 *
 * Features:
 * - Toggle rule active/inactive
 * - Currency selection dropdown
 * - Percentage or money amount input
 * - Real-time balance fetching from API
 * - Calculated amount display
 *
 * Relations:
 * - Store (NgRx): Reads and updates riskPerTrade configuration
 * - AppContextService: Gets report data and balance
 * - AuthService: Gets authentication tokens
 * - ReportService: Fetches account balance
 *
 * @component
 * @selector app-risk-per-trade
 * @standalone true
 */
@Component({
  selector: 'app-risk-per-trade',
  templateUrl: './risk-per-trade.component.html',
  styleUrls: ['./risk-per-trade.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class RiskPerTradeComponent implements OnInit {
  config: RiskPerTradeConfig = {
    isActive: false,
    review_type: 'MAX',
    number_type: 'PERCENTAGE',
    percentage_type: 'NULL',
    risk_ammount: 0,
    type: RuleType.MAX_RISK_PER_TRADE,
    balance: 1,
    actualBalance: 0,
  };

  actualBalance: number = 0;
  calculatedAmount: number = 0;
  inputFirstRatioValue: number = 0;

  inputSecondRatioValue: number = 0;

  initialRiskTrade: number | undefined;
```

```
  selectedCurrency = currencies[0];

  dropdownOpen = false;

  currencies = currencies;

  constructor(
    private store: Store,
    private settingsService: SettingsService,
    private appContext: AppContextService,
    private authService: AuthService,
    private reportService: ReportService
  ) {}

  closeDropdown() {
    this.dropdownOpen = false;
  }

  async ngOnInit(): Promise<void> {
    this.listenRuleConfiguration();
    await this.loadData();
  }

  async loadData() {
    await this.loadActualBalance();
    this.calculatedAmount = await this.getCalculatedAmount();
  }

  toggleDropdown() {
    if (this.config.isActive) {
      this.dropdownOpen = !this.dropdownOpen;
    } else {
      this.dropdownOpen = false;
    }
  }

  selectCurrency(currency: { code: string; country: string }) {
    this.selectedCurrency = currency;
    this.dropdownOpen = false;
  }

  onToggleActive(event: Event) {
    const newConfig = {
      ...this.config,
      isActive: (event.target as HTMLInputElement).checked,
    };
    this.updateConfig(newConfig);
  }

  onChangePercentage(event: Event) {
    const percentage = Number((event.target as HTMLInputElement).value);
    this.actualBalance = this.appContext.reportData()?.balanceData?.balance || 0;
    const moneyRisk = Number(
      ((percentage / 100) * this.actualBalance).toFixed(2)
    );

    const newConfig: RiskPerTradeConfig = {
      ...this.config,
      risk_ammount: percentage,
      balance: -1,
      actualBalance: this.actualBalance,
    };
    this.updateConfig(newConfig);
  }

  async getCurrentBalance(): Promise<number> {
    if (this.config.percentage_type === 'ACTUAL_B') {
      // Usar el actualBalance guardado si está disponible, sino cargar
      if (this.config.actualBalance && this.config.actualBalance > 0) {
```

```
        return this.config.actualBalance;
      } else if (this.actualBalance === 0) {
        await this.loadActualBalance();
        return this.actualBalance;
      }
      return this.actualBalance;
    } else if (this.config.percentage_type === 'INITIAL_B') {
      return this.config.balance;
    }
    return 0;
  }

  // Método para cargar el balance actual desde el servicio
  async loadActualBalance() {
    try {
      // Obtener datos del usuario para hacer la petición
      const currentUser = this.appContext.currentUser();
      if (!currentUser) {
        console.warn('No hay usuario autenticado');
        this.actualBalance = 0;
        return;
      }

      // Obtener la primera cuenta del usuario
      const userAccounts = this.appContext.userAccounts();
      if (!userAccounts || userAccounts.length === 0) {
        console.warn('No hay cuentas disponibles');
        this.actualBalance = 0;
        return;
      }

      const account = userAccounts[0];
      const accessToken = await this.authService.getBearerTokenFirebase(currentUser.id);

      // Hacer la petición al servicio de reportes para obtener el balance
      const balanceData = await this.reportService.getBalanceData(
        account.accountID,
        accessToken,
        account.accountNumber
      ).toPromise();

      if (balanceData && balanceData.balance) {
        this.actualBalance = balanceData.balance;
        // Actualizar el contexto con los datos obtenidos
        this.appContext.updateReportBalance(balanceData);
      } else {
        this.actualBalance = 0;
      }
    } catch (error) {
      console.error('Error loading actual balance:', error);
      this.actualBalance = 0;
    }
  }

  async getCalculatedAmount(): Promise<number> {
    const balance = await this.getCurrentBalance();
    return (this.config.risk_ammount / 100) * balance;
  }

  onChangeAmount(event: Event) {
    const moneyRisk = Number((event.target as HTMLInputElement).value);
    this.actualBalance = this.appContext.reportData()?.balanceData?.balance || 0;
    const percentage = Number(
      ((moneyRisk / this.actualBalance) * 100).toFixed(2)
    );

    const newConfig: RiskPerTradeConfig = {
      ...this.config,
      risk_ammount: moneyRisk,
      balance: 0,
```

```
      actualBalance: 0,
    };
    this.updateConfig(newConfig);
  }

  listenRuleConfiguration() {
    this.store
      .select(riskPerTrade)
      .pipe()
      .subscribe((config) => {
        this.config = config;
        if (!this.initialRiskTrade) {
          this.initialRiskTrade = config.risk_ammount;
        }

        // Si la regla está inactiva, resetear todos los valores
        if (!config.isActive) {
          this.actualBalance = 0;
          this.calculatedAmount = 0;
        } else {
          // Si está activa, cargar actualBalance si está disponible
          if (config.actualBalance && config.actualBalance > 0) {
            this.actualBalance = config.actualBalance;
          }
        }
      });
  }

  private updateConfig(config: RiskPerTradeConfig) {
    this.store.dispatch(setRiskPerTradeConfig({ config }));
  }
}

```

Ø=ÜÁ features\strategy\components\risk-per-trade\models

Ø=ÜÄ features\strategy\components\risk-per-trade\models\risk-per-trade.model.ts

```
export const currencies = [
  { code: 'USD', country: 'US' },
  { code: 'EUR', country: 'EU' },
  { code: 'JPY', country: 'JP' },
  { code: 'GBP', country: 'GB' },
  { code: 'AUD', country: 'AU' },
  { code: 'CAD', country: 'CA' },
  { code: 'CHF', country: 'CH' },
  { code: 'CNY', country: 'CN' },
  { code: 'HKD', country: 'HK' },
  { code: 'NZD', country: 'NZ' },
];

```

Ø=ÜÁ features\strategy\components\risk-reward

Ø=ÜÄ features\strategy\components\risk-reward\risk-reward.component.ts

```ts
import { CommonModule } from '@angular/common';
import { Component, OnInit } from '@angular/core';
import {
  MaxDailyTradesConfig,
  RiskRewardConfig,
  RuleType,
} from '../../models/strategy.model';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../service/strategy.service';
import { riskReward } from '../../store/strategy.selectors';
import { setRiskRewardConfig } from '../../store/strategy.actions';

/**
 * Component for configuring the risk/reward ratio rule.
 *
 * This component allows users to set a minimum risk/reward ratio (e.g., "1:2")
 * that trades must meet. It provides input fields for both parts of the ratio
 * and includes increment/decrement buttons for the second value.
 *
 * Features:
 * - Toggle rule active/inactive
 * - Input fields for risk and reward values
 * - Increment/decrement buttons for reward value
 * - Syncs with NgRx store
 *
 * Relations:
 * - Store (NgRx): Reads and updates riskReward configuration
 * - SettingsService: Strategy service (injected but not directly used)
 *
 * @component
 * @selector app-risk-reward-ratio
 * @standalone true
 */
@Component({
  selector: 'app-risk-reward-ratio',
  templateUrl: './risk-reward.component.html',
  styleUrls: ['./risk-reward.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class RiskRewardComponent implements OnInit {
  config: RiskRewardConfig = {
    isActive: false,
    riskRewardRatio: '1:2',
    type: RuleType.RISK_REWARD_RATIO,
  };

  inputFirstRatioValue: number = 0;

  inputSecondRatioValue: number = 0;

  initialRatio: string | undefined;

  constructor(private store: Store, private settingsService: SettingsService) {}

  ngOnInit(): void {
    this.listenRuleConfiguration();
  }
  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Reiniciar a 1:2 cuando se desactiva
      riskRewardRatio: isActive ? this.config.riskRewardRatio : '1:2',
    };
    this.updateConfig(newConfig);
  }

  onChangeValue(event: Event, isFirst: boolean) {
```

```
    const numValue = Number((event.target as HTMLInputElement).value);
    const numberArray = this.config.riskRewardRatio
      .split(':')
      .map((number) => parseInt(number, 10));

    if (isFirst) {
      numberArray[0] = numValue;
    } else {
      numberArray[1] = numValue;
    }

    const newConfig: RiskRewardConfig = {
      ...this.config,
      riskRewardRatio: numberArray.join(':'),
    };
    this.updateConfig(newConfig);
  }

  // Métodos para spinner (solo para el segundo número)
  incrementSecondValue() {
    if (this.config.isActive) {
      const numberArray = this.config.riskRewardRatio
        .split(':')
        .map((number) => parseInt(number, 10));

      numberArray[1] = numberArray[1] + 1;

      const newConfig: RiskRewardConfig = {
        ...this.config,
        riskRewardRatio: numberArray.join(':'),
      };
      this.updateConfig(newConfig);
    }
  }

  decrementSecondValue() {
    if (this.config.isActive) {
      const numberArray = this.config.riskRewardRatio
        .split(':')
        .map((number) => parseInt(number, 10));

      if (numberArray[1] > 2) {
        numberArray[1] = numberArray[1] - 1;

        const newConfig: RiskRewardConfig = {
          ...this.config,
          riskRewardRatio: numberArray.join(':'),
        };
        this.updateConfig(newConfig);
      }
    }
  }

  listenRuleConfiguration() {
    this.store
      .select(riskReward)
      .pipe()
      .subscribe((config) => {
        this.config = config;
        if (!this.initialRatio) {
          this.initialRatio = config.riskRewardRatio;
        }
        const numberArray = config.riskRewardRatio
          .split(':')
          .map((number) => parseInt(number, 10));
        this.inputFirstRatioValue = numberArray[0];
        this.inputSecondRatioValue = numberArray[1];
      });
  }

```

```
  private updateConfig(config: RiskRewardConfig) {
    this.store.dispatch(setRiskRewardConfig({ config }));
  }
}

```

Ø=ÜÁ features\strategy\edit-strategy

Ø=ÜÄ features\strategy\edit-strategy\edit-strategy.component.ts

```
import { Component, OnInit, OnDestroy, ViewChild } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { FormsModule } from '@angular/forms';
import { NgIf } from '@angular/common';
import { allRules } from '../store/strategy.selectors';
import { StrategyState, RuleType } from '../models/strategy.model';
import { resetConfig } from '../store/strategy.actions';
import { SettingsService } from '../service/strategy.service';
import { ReportService } from '../../report/service/report.service';
import { User } from '../../overview/models/overview';
import { selectUser } from '../../auth/store/user.selectios';
import { selectUserKey } from '../../report/store/report.selectors';
import { setUserKey } from '../../report/store/report.actions';
import { initialStrategyState } from '../store/strategy.reducer';
import { EditPopupComponent } from '../../../shared/pop-ups/edit-pop-up/edit-popup.component';
import { ConfirmPopupComponent } from '../../../shared/pop-ups/confirm-pop-up/confirm-popup.component';
import { LoadingPopupComponent } from '../../../shared/pop-ups/loading-pop-up/loading-popup.component';
// Importar componentes de reglas edit-*
import { EditMaxDailyTradesComponent } from './components/edit-max-daily-trades/edit-max-daily-trades.component';
import { EditRiskRewardComponent } from './components/edit-risk-reward/edit-risk-reward.component';
import { EditRiskPerTradeComponent } from './components/edit-risk-per-trade/edit-risk-per-trade.component';
import { EditDaysAllowedComponent } from './components/edit-days-allowed/edit-days-allowed.component';
import { EditAssetsAllowedComponent } from './components/edit-assets-allowed/edit-assets-allowed.component';
import { EditHoursAllowedComponent } from './components/edit-hours-allowed/edit-hours-allowed.component';
import { AuthService } from '../../auth/service/authService';
import { AccountData } from '../../auth/models/userModel';
import { PluginHistoryService, PluginHistory } from '../../../shared/services/plugin-history.service';
import { AlertService } from '../../../shared/services/alert.service';
import { Instrument } from '../../report/models/report.model';
import { StrategyCacheService } from '../services/strategy-cache.service';
import { BalanceCacheService } from '../services/balance-cache.service';

/**
 * Component for editing trading strategy configurations.
 *
 * This component provides a comprehensive interface for editing all aspects
 * of a trading strategy, including all six trading rules. It supports both
 * creating new strategies and editing existing ones.
 *
 * Features:
 * - Edit all six trading rules (risk/reward, risk per trade, max daily trades,
 *   days allowed, hours allowed, assets allowed)
 * - Strategy name editing
 * - Save and cancel functionality
 * - Validation before saving
 * - Loading states
 * - Plugin history integration
 * - Instrument fetching for assets allowed
 *
 * Relations:
```

```
 * - EditMaxDailyTradesComponent: Max daily trades rule editor
 * - EditRiskRewardComponent: Risk/reward ratio rule editor
 * - EditRiskPerTradeComponent: Risk per trade rule editor
 * - EditDaysAllowedComponent: Days allowed rule editor
 * - EditHoursAllowedComponent: Trading hours rule editor
 * - EditAssetsAllowedComponent: Assets allowed rule editor
 * - SettingsService: Saving strategy configurations
 * - StrategyCacheService: Loading strategies from cache
 * - BalanceCacheService: Getting account balances
 * - Store (NgRx): Managing strategy state
 *
 * @component
 * @selector app-edit-strategy
 * @standalone true
 */
@Component({
  selector: 'app-edit-strategy',
  imports: [
    FormsModule,
    NgIf,
    EditMaxDailyTradesComponent,
    EditRiskRewardComponent,
    EditRiskPerTradeComponent,
    EditDaysAllowedComponent,
    EditAssetsAllowedComponent,
    EditHoursAllowedComponent,
    EditPopupComponent,
    ConfirmPopupComponent,
    LoadingPopupComponent
  ],
  templateUrl: './edit-strategy.component.html',
  styleUrl: './edit-strategy.component.scss',
  standalone: true,
})
export class EditStrategyComponent implements OnInit, OnDestroy {
  config$: Observable<StrategyState>;
  config: StrategyState | null = null;
  myChoices: StrategyState | null = null;
  loading = false;
  initialLoading = true; // Loading global para evitar tambaleo
  user: User | null = null;
  editPopupVisible = false;
  confirmPopupVisible = false;
  strategyId: string | null = null;

  // Mini card properties
  currentStrategyName: string = 'My Strategy';
  lastModifiedText: string = 'Never modified';
  isFavorited: boolean = false;
  isEditingName: boolean = false;
  editingStrategyName: string = '';
  accountsData: AccountData[] = [];
  currentAccount: AccountData | null = null;
  availableInstruments: string[] = [];

  // Plugin history properties
  pluginHistory: PluginHistory[] = [];
  isPluginActive: boolean = false;
  private pluginSubscription: any = null;

  // Referencias a componentes de reglas para validación
  @ViewChild('hoursAllowedRef') hoursAllowedRef?: EditHoursAllowedComponent;
  @ViewChild('daysAllowedRef') daysAllowedRef?: EditDaysAllowedComponent;
  @ViewChild('assetsAllowedRef') assetsAllowedRef?: EditAssetsAllowedComponent;
  @ViewChild('maxDailyTradesRef') maxDailyTradesRef?: EditMaxDailyTradesComponent;
  @ViewChild('riskPerTradeRef') riskPerTradeRef?: EditRiskPerTradeComponent;

  constructor(
    private store: Store,
    private router: Router,
```

```
    private route: ActivatedRoute,
    private strategySvc: SettingsService,
    private reportSvc: ReportService,
    private authService: AuthService,
    private pluginHistoryService: PluginHistoryService,
    private strategyCacheService: StrategyCacheService,
    private balanceCacheService: BalanceCacheService,
    private alertService: AlertService
  ) {
    this.config$ = this.store.select(allRules);
  }

  async ngOnInit() {
    this.initialLoading = true;

    try {
      // FLUJO SIMPLIFICADO: Cargar todo antes de mostrar la UI
      await this.initializeEverything();
    } catch (error) {
      console.error('Error during initialization:', error);
    } finally {
      this.initialLoading = false;
    }
  }

  /**
   * Inicializar todo antes de mostrar la UI
   */
  private async initializeEverything(): Promise<void> {
    // 1. Obtener datos del usuario
    await this.getUserDataAsync();

    // 2. Configurar listeners
    this.listenConfigurations();

    // 3. Obtener ID de estrategia y cargar configuración
    this.getStrategyId();
  }

  /**
   * MÉTODO SIMPLIFICADO: Obtener ID de estrategia desde la URL
   * - Si hay strategyId: Cargar desde cache
   * - Si no hay strategyId: Nueva estrategia
   */
  getStrategyId() {
    this.route.queryParams.subscribe(params => {
      const newStrategyId = params['strategyId'] || null;

      // Si cambió la estrategia, limpiar el estado
      if (this.strategyId !== newStrategyId) {
        this.clearState();
      }

      this.strategyId = newStrategyId;

      // Cargar configuración usando el cache
      this.loadStrategyFromCache();
    });
  }

  /**
   * Limpia el estado al cambiar entre estrategias
   */
  clearState() {
    // Limpiar variables del componente
    this.myChoices = null;
    this.config = null;
    this.currentStrategyName = 'My Strategy';
    this.lastModifiedText = 'Never modified';
    this.isFavorited = false;
```

```
    this.isEditingName = false;
    this.editingStrategyName = '';

    // LIMPIAR EL STORE para evitar datos de estrategias anteriores
    // Usar un estado inicial vacío en lugar de null
    const emptyState: StrategyState = {
      maxDailyTrades: { isActive: false, type: RuleType.MAX_DAILY_TRADES, maxDailyTrades: 0 },
      riskReward: { isActive: false, type: RuleType.RISK_REWARD_RATIO, riskRewardRatio: '1:2' },
      riskPerTrade: { isActive: false, type: RuleType.MAX_RISK_PER_TRADE, review_type: 'MAX', number_type: 'PERCENTAGE', percentage_type: 'NULL', risk_ammount: 0, balance: 0, actualBalance: 0 },
      daysAllowed: { isActive: false, type: RuleType.DAYS_ALLOWED, tradingDays: [] },
      assetsAllowed: { isActive: false, type: RuleType.ASSETS_ALLOWED, assetsAllowed: [] },
      hoursAllowed: { isActive: false, type: RuleType.TRADING_HOURS, tradingOpenTime: '', tradingCloseTime: '', timezone: '' }
    };

    this.store.dispatch(resetConfig({ config: emptyState }));
  }

  /**
   * MÉTODO PRINCIPAL: Cargar estrategia desde cache
   * FLUJO SIMPLIFICADO:
   * - Si hay strategyId: Cargar desde cache del componente Strategy
   * - Si no hay strategyId: Inicializar como nueva estrategia
   */
  loadStrategyFromCache() {
    if (this.strategyId) {
      // Cargar estrategia existente desde cache
      this.loadExistingStrategyFromCache();
    } else {
      // Inicializar como nueva estrategia
      this.initializeAsNewStrategy();
    }
  }

  /**
   * Cargar estrategia existente desde cache o Firebase
   */
  loadExistingStrategyFromCache() {
    if (!this.strategyId) return;

    // Verificar si el cache está disponible
    if (!this.strategyCacheService.isCacheLoaded()) {
      // Si el cache no está disponible, cargar directamente desde Firebase
      this.loadStrategyFromFirebase();
      return;
    }

    // Obtener estrategia del cache
    const cachedStrategy = this.strategyCacheService.getStrategy(this.strategyId);

    if (!cachedStrategy) {
      // Si no está en cache, cargar desde Firebase
      this.loadStrategyFromFirebase();
      return;
    }

    // Actualizar mini card
    this.currentStrategyName = cachedStrategy.overview.name;
    this.lastModifiedText = this.formatDate(cachedStrategy.overview.updated_at.toDate());
    this.isFavorited = false;

    // Cargar balance si es necesario
    this.loadBalanceAndInitializeWithConfig(cachedStrategy.configuration);
  }

  /**
   * Cargar estrategia directamente desde Firebase (fallback cuando cache no está disponible)
   */
  async loadStrategyFromFirebase() {
    if (!this.strategyId || !this.user?.id) {
      this.initializeAsNewStrategy();
```

```
      return;
    }

    try {
      // Cargar estrategia directamente desde Firebase
      const strategyData = await this.strategySvc.getStrategyView(this.strategyId);

      if (!strategyData || !strategyData.configuration) {
        this.initializeAsNewStrategy();
        return;
      }

      // Actualizar mini card
      this.currentStrategyName = strategyData.overview.name;
      this.lastModifiedText = this.formatDate(strategyData.overview.updated_at.toDate());
      this.isFavorited = false;

      // Cargar balance y inicializar con configuración
      this.loadBalanceAndInitializeWithConfig(strategyData.configuration);

    } catch (error) {
      console.error('Error loading strategy from Firebase:', error);
      this.initializeAsNewStrategy();
    }
  }

  /**
   * Cargar balance e inicializar con configuración específica
   */
  loadBalanceAndInitializeWithConfig(configuration: StrategyState) {
    // Obtener balance desde cache primero
    let balance = 0;
    if (this.accountsData.length > 0) {
      const firstAccount = this.accountsData[0];
      balance = this.balanceCacheService.getBalance(firstAccount.accountID);
    }

    // Inicializar con balance del cache
    this.initializeWithConfigAndBalance(configuration, balance);

    // Si necesita actualización, hacer petición en background
    if (this.accountsData.length > 0) {
      const firstAccount = this.accountsData[0];
      if (this.balanceCacheService.needsUpdate(firstAccount.accountID)) {
        this.updateBalanceInBackground(firstAccount);
      }
    }
  }

  /**
   * Actualizar balance en background sin bloquear la UI
   */
  private updateBalanceInBackground(account: AccountData) {
    this.store.select(selectUserKey).pipe().subscribe({
      next: (userKey) => {
        if (userKey && userKey !== '') {
          this.reportSvc.getBalanceData(account.accountID as string, userKey,
account.accountNumber as number).subscribe({
            next: (balance) => {
              // Actualizar cache con nuevo balance
              this.balanceCacheService.setBalance(account.accountID, balance);
            },
            error: (err) => {
              console.error('Error fetching balance data in background:', err);
            },
          });
        }
      },
    });
  }

```

```
  /**
   * Inicializar con configuración y balance específicos
   */
  initializeWithConfigAndBalance(configuration: StrategyState, balance: number) {
    // Solo actualizar balance si es para balance actual (ACTUAL_B)
    // Para balance inicial (INITIAL_B), mantener el valor de Firebase
    let configWithBalance = { ...configuration };

    if (configuration.riskPerTrade.percentage_type === 'ACTUAL_B') {
      // Para balance actual, usar el balance del cache
      configWithBalance = {
        ...configuration,
        riskPerTrade: {
          ...configuration.riskPerTrade,
          actualBalance: balance
        }
      };
    } else if (configuration.riskPerTrade.percentage_type === 'INITIAL_B') {
      // Para balance inicial, mantener el valor de Firebase (no modificar)
      configWithBalance = configuration;
    } else {
      // Para otros casos, usar el balance del cache
      configWithBalance = {
        ...configuration,
        riskPerTrade: {
          ...configuration.riskPerTrade,
          balance: balance
        }
      };
    }

    // Cargar en el store
    this.store.dispatch(resetConfig({ config: configWithBalance }));

    // Distribuir reglas entre paneles
    this.distributeRulesBetweenPanels(configWithBalance);
  }

  /**
   * MÉTODO SIMPLIFICADO: Inicializar como nueva estrategia
   * FLUJO PARA NUEVA ESTRATEGIA:
   * - Available Rules: Todas las reglas (isActive = true para mostrar)
   * - My Choices: Vacío (isActive = false)
   */
  initializeAsNewStrategy() {
    // Cargar configuración por defecto con balance
    this.loadBalanceAndInitialize();
  }

  /**
   * MÉTODO SIMPLIFICADO: Distribuir reglas entre paneles
   * LÓGICA UNIFICADA:
   * - My Choices: Reglas con isActive = true
   * - Available Rules: Reglas con isActive = false
   */
  distributeRulesBetweenPanels(configurationData: StrategyState) {
    // My Choices: Solo reglas activas (isActive = true)
    this.myChoices = {
      maxDailyTrades: {
        isActive: configurationData.maxDailyTrades.isActive,
        type: configurationData.maxDailyTrades.type,
        maxDailyTrades: configurationData.maxDailyTrades.maxDailyTrades
      },
      riskReward: {
        isActive: configurationData.riskReward.isActive,
        type: configurationData.riskReward.type,
        riskRewardRatio: configurationData.riskReward.riskRewardRatio
      },
      riskPerTrade: {
        isActive: configurationData.riskPerTrade.isActive,
```

```
        type: configurationData.riskPerTrade.type,
        review_type: configurationData.riskPerTrade.review_type,
        number_type: configurationData.riskPerTrade.number_type,
        percentage_type: configurationData.riskPerTrade.percentage_type,
        risk_ammount: configurationData.riskPerTrade.risk_ammount,
        balance: configurationData.riskPerTrade.balance,
        actualBalance: configurationData.riskPerTrade.actualBalance
      },
      daysAllowed: {
        isActive: configurationData.daysAllowed.isActive,
        type: configurationData.daysAllowed.type,
        tradingDays: configurationData.daysAllowed.tradingDays
      },
      assetsAllowed: {
        isActive: configurationData.assetsAllowed.isActive,
        type: configurationData.assetsAllowed.type,
        assetsAllowed: configurationData.assetsAllowed.assetsAllowed
      },
      hoursAllowed: {
        isActive: configurationData.hoursAllowed.isActive,
        type: configurationData.hoursAllowed.type,
        tradingOpenTime: configurationData.hoursAllowed.tradingOpenTime,
        tradingCloseTime: configurationData.hoursAllowed.tradingCloseTime,
        timezone: configurationData.hoursAllowed.timezone
      }
    };

    // Available Rules: Solo reglas NO activas (isActive = false)
    this.config = {
      maxDailyTrades: {
        isActive: !configurationData.maxDailyTrades.isActive,
        type: configurationData.maxDailyTrades.type,
        maxDailyTrades: configurationData.maxDailyTrades.maxDailyTrades
      },
      riskReward: {
        isActive: !configurationData.riskReward.isActive,
        type: configurationData.riskReward.type,
        riskRewardRatio: configurationData.riskReward.riskRewardRatio
      },
      riskPerTrade: {
        isActive: !configurationData.riskPerTrade.isActive,
        type: configurationData.riskPerTrade.type,
        review_type: configurationData.riskPerTrade.review_type,
        number_type: configurationData.riskPerTrade.number_type,
        percentage_type: configurationData.riskPerTrade.percentage_type,
        risk_ammount: configurationData.riskPerTrade.risk_ammount,
        balance: configurationData.riskPerTrade.balance,
        actualBalance: configurationData.riskPerTrade.actualBalance
      },
      daysAllowed: {
        isActive: !configurationData.daysAllowed.isActive,
        type: configurationData.daysAllowed.type,
        tradingDays: configurationData.daysAllowed.tradingDays
      },
      assetsAllowed: {
        isActive: !configurationData.assetsAllowed.isActive,
        type: configurationData.assetsAllowed.type,
        assetsAllowed: configurationData.assetsAllowed.assetsAllowed
      },
      hoursAllowed: {
        isActive: !configurationData.hoursAllowed.isActive,
        type: configurationData.hoursAllowed.type,
        tradingOpenTime: configurationData.hoursAllowed.tradingOpenTime,
        tradingCloseTime: configurationData.hoursAllowed.tradingCloseTime,
        timezone: configurationData.hoursAllowed.timezone
      }
    };
  }

  /**
```

```
   * Cargar cuentas de forma asíncrona
   */
  private async fetchUserAccountsAsync(): Promise<void> {
    if (this.user?.id) {
      try {
        const accounts = await this.authService.getUserAccounts(this.user.id);
        this.accountsData = accounts || [];
        // After loading accounts, try to fetch user key
        await this.fetchUserKeyAsync();
      } catch (error) {
        console.error('Error loading accounts:', error);
        this.accountsData = [];
      }
    }
  }

  fetchUserAccounts() {
    if (this.user?.id) {
      this.authService.getUserAccounts(this.user.id).then((accounts) => {
        this.accountsData = accounts || [];
        // After loading accounts, try to fetch user key
        this.fetchUserKey();
      });
    }
  }

  /**
   * Obtener userKey de forma asíncrona
   */
  private async fetchUserKeyAsync(): Promise<void> {
    if (this.user?.email && this.accountsData.length > 0) {
      // Use the first account's credentials
      const firstAccount = this.accountsData[0];

      try {
        const key = await this.reportSvc.getUserKey(
          firstAccount.emailTradingAccount,
          firstAccount.brokerPassword,
          firstAccount.server
        ).toPromise();

        this.store.dispatch(setUserKey({ userKey: key || '' }));
        // After getting userKey, load instruments
        this.loadInstruments(key || '', firstAccount);
      } catch (err) {
        console.error('Error fetching user key:', err);
        this.store.dispatch(setUserKey({ userKey: '' }));
      }
    } else {
      this.store.dispatch(setUserKey({ userKey: '' }));
    }
  }

  fetchUserKey() {
    if (this.user?.email && this.accountsData.length > 0) {
      // Use the first account's credentials
      const firstAccount = this.accountsData[0];
      this.reportSvc
        .getUserKey(firstAccount.emailTradingAccount, firstAccount.brokerPassword,
firstAccount.server)
        .subscribe({
          next: (key: string) => {
            this.store.dispatch(setUserKey({ userKey: key }));
            // After getting userKey, load instruments
            this.loadInstruments(key, firstAccount);
          },
          error: (err) => {
            console.error('Error fetching user key:', err);
            this.store.dispatch(setUserKey({ userKey: '' }));
          },
        });
```

```
        } else {
          this.store.dispatch(setUserKey({ userKey: '' }));
        }
  }

  loadInstruments(userKey: string, account: AccountData) {
    this.reportSvc.getAllInstruments(
      userKey,
      account.accountNumber,
      account.accountID
    ).subscribe({
      next: (instruments: Instrument[]) => {
        // Extraer solo los nombres de los instrumentos
        this.availableInstruments = instruments.map(instrument => instrument.name);
      },
      error: (err) => {
        console.error('Error loading instruments:', err);
        this.availableInstruments = [];
      }
    });
  }

  getActualBalance() {
    this.store
      .select(selectUserKey)
      .pipe()
      .subscribe({
        next: (userKey) => {
          if (userKey === '') {
            this.fetchUserKey();
          } else {
            // Use the first account's data dynamically
            if (this.accountsData.length > 0) {
              const firstAccount = this.accountsData[0];
              // El servicio ya actualiza el contexto automáticamente
              this.reportSvc.getBalanceData(firstAccount.accountID as string, userKey,
firstAccount.accountNumber as number).subscribe({
                next: (balance) => {
                  this.initializeWithConfigAndBalance(initialStrategyState, balance);
                },
                error: (err) => {
                  console.error('Error fetching balance data', err);
                  this.initializeWithConfigAndBalance(initialStrategyState, 0);
                },
              });
            } else {
              this.initializeWithConfigAndBalance(initialStrategyState, 0);
            }
          }
        },
      });
  }

  /**
   * Obtener datos del usuario de forma asíncrona
   */
  private async getUserDataAsync(): Promise<void> {
    return new Promise((resolve) => {
      this.store.select(selectUser).subscribe({
        next: async (user) => {
          this.user = user.user;

          // Cargar cuentas y configurar plugin history cuando el usuario esté disponible
          if (this.user?.id) {
            await this.fetchUserAccountsAsync();
            this.setupPluginHistoryListener();
          } else {
            console.warn('User ID not available yet');
          }
          resolve();
        },
```

```
        error: (err) => {
          console.error('Error fetching user data', err);
          resolve();
        },
      });
    });
  }

  getUserData() {
    this.store.select(selectUser).subscribe({
      next: (user) => {
        this.user = user.user;

        // Cargar cuentas y configurar plugin history cuando el usuario esté disponible
        if (this.user?.id) {
          this.fetchUserAccounts();
          this.setupPluginHistoryListener();
        } else {
          console.warn('User ID not available yet');
        }
      },
      error: (err) => {
        console.error('Error fetching user data', err);
      },
    });
  }

  /**
   * MÉTODO SIMPLIFICADO: Cargar balance e inicializar como nueva estrategia
   */
  loadBalanceAndInitialize() {
    // Obtener balance desde cache primero
    let balance = 0;
    if (this.accountsData.length > 0) {
      const firstAccount = this.accountsData[0];
      balance = this.balanceCacheService.getBalance(firstAccount.accountID);
    }

    // Inicializar con balance del cache
    this.initializeWithConfigAndBalance(initialStrategyState, balance);

    // Si necesita actualización, hacer petición en background
    if (this.accountsData.length > 0) {
      const firstAccount = this.accountsData[0];
      if (this.balanceCacheService.needsUpdate(firstAccount.accountID)) {
        this.updateBalanceInBackground(firstAccount);
      }
    }
  }

  /**
   * MÉTODO SIMPLIFICADO: Escuchar cambios en el store de reglas
   */
  listenConfigurations() {
    this.store
      .select(allRules)
      .pipe()
      .subscribe((config) => {
        // Actualizar UI en tiempo real
        this.listenToStoreChanges();
      });
  }


  /**
   * MÉTODO SIMPLIFICADO: Escuchar cambios en tiempo real
   */
  listenToStoreChanges() {
    this.config$.subscribe(config => {
      if (this.config && this.myChoices) {
```

```
        // Actualizar UI en tiempo real
        this.updateMyChoicesFromConfig(config);
      }
    });
  }

  /**
   * MÉTODO SIMPLIFICADO: Actualizar UI en tiempo real
   */
  updateMyChoicesFromConfig(newConfig: StrategyState) {
    if (!this.myChoices || !this.config) return;

    // Usar el método unificado para distribuir reglas
    this.distributeRulesBetweenPanels(newConfig);
  }

  saveStrategy() {
    // Verificar si se puede guardar (plugin no activo)
    if (!this.canSaveStrategy()) {
      this.alertService.showWarning('Cannot save strategy while plugin is active. Please deactivate the plugin first.', 'Plugin Active');
      return;
    }

    // Validar todas las reglas activas
    if (!this.validateActiveRules()) {
      return;
    }

    this.confirmPopupVisible = true;
  }

  /**
   * MÉTODO 7: Guardar configuración
   * FLUJO DE GUARDADO:
   * - Solo se guardan las reglas que están en My Choices (isActive = true)
   * - Si hay strategyId: Actualizar estrategia existente
   * - Si no hay strategyId: Crear nueva estrategia
   */
  save = () => {
    if (this.myChoices && this.user?.id) {
      this.loading = true;

      // GUARDADO: Crear configuración solo con las reglas de My Choices
      // Las reglas que están en My Choices son las que se guardarán como activas
      const newConfig: StrategyState = {
        maxDailyTrades: { isActive: this.myChoices.maxDailyTrades.isActive, type: this.myChoices.maxDailyTrades.type, maxDailyTrades: this.myChoices.maxDailyTrades.maxDailyTrades },
        riskReward: { isActive: this.myChoices.riskReward.isActive, type: this.myChoices.riskReward.type, riskRewardRatio: this.myChoices.riskReward.riskRewardRatio },
        riskPerTrade: { isActive: this.myChoices.riskPerTrade.isActive, type: this.myChoices.riskPerTrade.type, review_type: this.myChoices.riskPerTrade.review_type, number_type: this.myChoices.riskPerTrade.number_type, percentage_type: this.myChoices.riskPerTrade.percentage_type, risk_ammount: this.myChoices.riskPerTrade.risk_ammount, actual_balance: this.myChoices.riskPerTrade.actual_balance },
        daysAllowed: { isActive: this.myChoices.daysAllowed.isActive, type: this.myChoices.daysAllowed.type, tradingDays: this.myChoices.daysAllowed.tradingDays },
        hoursAllowed: { isActive: this.myChoices.hoursAllowed.isActive, type: this.myChoices.hoursAllowed.type, tradingOpenTime: this.myChoices.hoursAllowed.tradingOpenTime, tradingCloseTime: this.myChoices.hoursAllowed.tradingCloseTime, timezone: this.myChoices.hoursAllowed.timezone }
      };
      // Actualizar el store con la nueva configuración
      this.store.dispatch(resetConfig({ config: newConfig }));

      // CASO A: Estrategia existente - Actualizar en Firebase
      if (this.strategyId) {
        this.strategySvc
          .updateStrategyView(this.strategyId, {
            configuration: newConfig
          })
          .then(async () => {
            // Actualizar fecha de modificación en la mini card
            this.lastModifiedText = this.formatDate(new Date());

            // Limpiar cache para forzar recarga
            this.strategyCacheService.clearCache();

            this.router.navigate(['/strategy']);
```

```
          })
          .catch((err) => {
            console.error('Update Error:', err);
            this.alertService.showError('Error Updating Strategy', 'Update Error');
          })
          .finally(() => {
            this.loading = false;
            this.closeConfirmModal();
          });
      } else {
        // CASO B: Nueva estrategia - Crear en Firebase
        this.strategySvc
          .createStrategyView(this.user.id, this.currentStrategyName, newConfig)
          .then(async (strategyId) => {
            // Limpiar cache para forzar recarga
            this.strategyCacheService.clearCache();

            this.router.navigate(['/strategy']);
          })
          .catch((err) => {
            console.error('Create Error:', err);
            this.alertService.showError('Error Creating Strategy', 'Creation Error');
          })
          .finally(() => {
            this.loading = false;
            this.closeConfirmModal();
          });
      }
    }
  };

  closePopup = () => {
    this.editPopupVisible = false;
  };

  openConfirmPopup() {
    this.confirmPopupVisible = true;
  }

  closeConfirmModal = () => {
    this.confirmPopupVisible = false;
  };

  /**
   * Método que se ejecuta cuando se confirma el guardado
   * Hace la validación final antes de proceder con el guardado
   */
  confirmSave = () => {
    // Validar todas las reglas activas una vez más
    if (!this.validateActiveRules()) {
      return; // No proceder con el guardado
    }

    this.save();
  };

  /**
   * Valida todas las reglas activas antes de permitir guardar
   */
  private validateActiveRules(): boolean {
    if (!this.myChoices) {
      return true; // Si no hay reglas activas, permitir guardar
    }

    const validationErrors: string[] = [];

    // Validar regla de horas permitidas
    if (this.myChoices.hoursAllowed.isActive && this.hoursAllowedRef) {
      if (!this.hoursAllowedRef.isRuleValid()) {
        validationErrors.push(`Hours Allowed: ${this.hoursAllowedRef.getErrorMessage()}`);
```

```
      }
    }

    // Validar regla de días permitidos
    if (this.myChoices.daysAllowed.isActive && this.daysAllowedRef) {
      if (!this.daysAllowedRef.isRuleValid()) {
        validationErrors.push(`Days Allowed: ${this.daysAllowedRef.getErrorMessage()}`);
      }
    }

    // Validar regla de assets permitidos
    if (this.myChoices.assetsAllowed.isActive && this.assetsAllowedRef) {
      if (!this.assetsAllowedRef.isRuleValid()) {
        validationErrors.push(`Assets Allowed: ${this.assetsAllowedRef.getErrorMessage()}`);
      }
    }

    // Validar regla de máximo de trades diarios
    if (this.myChoices.maxDailyTrades.isActive && this.maxDailyTradesRef) {
      if (!this.maxDailyTradesRef.isRuleValid()) {
        validationErrors.push(`Max Daily Trades: ${this.maxDailyTradesRef.getErrorMessage()}
`);   }
    }

    // Validar regla de riesgo por trade
    if (this.myChoices.riskPerTrade.isActive && this.riskPerTradeRef) {
      if (!this.riskPerTradeRef.isRuleValid()) {
        validationErrors.push(`Risk Per Trade: ${this.riskPerTradeRef.getErrorMessage()}`);
      }
    }

    if (validationErrors.length > 0) {
      const errorMessage = 'Cannot save strategy. Please complete the following rules:\n\n'
+                          validationErrors.join('\n') +
                          '\n\nAll required fields must be filled before saving.';
      this.alertService.showError(errorMessage, 'Validation Error');
      return false;
    }

    return true;
  }

  // Mini card methods
  startEditName() {
    this.isEditingName = true;
    this.editingStrategyName = this.currentStrategyName;
    // Focus on input after view update
    setTimeout(() => {
      const input = document.querySelector('.strategy-name-input') as HTMLInputElement;
      if (input) {
        input.focus();
        input.select();
      }
    }, 0);
  }

  async saveStrategyName() {
    if (!this.editingStrategyName.trim()) {
      this.cancelEditName();
      return;
    }

    if (this.editingStrategyName.trim() === this.currentStrategyName) {
      this.cancelEditName();
      return;
    }

    try {
      if (this.strategyId) {
        // Actualizar nombre en configuration-overview
```

```
        await this.strategySvc.updateConfigurationOverview(this.strategyId, {
          name: this.editingStrategyName.trim()
        });

        // Actualizar UI
        this.currentStrategyName = this.editingStrategyName.trim();
        this.lastModifiedText = this.formatDate(new Date());
      } else {
        // Si no hay strategyId, solo actualizar localmente
        this.currentStrategyName = this.editingStrategyName.trim();
        this.lastModifiedText = this.formatDate(new Date());
      }

      this.isEditingName = false;
    } catch (error) {
      console.error('Error updating strategy name:', error);
      this.alertService.showError('Error updating strategy name', 'Name Update Error');
      this.cancelEditName();
    }
  }

  cancelEditName() {
    this.isEditingName = false;
    this.editingStrategyName = '';
  }

  toggleFavorite() {
    this.isFavorited = !this.isFavorited;
    // TODO: Implementar persistencia de favoritos en Firebase
    console.log('Strategy favorited:', this.isFavorited);
  }

  formatDate(date: Date): string {
    const months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
'Nov', 'Dec'];
    const month = months[date.getMonth()];
    const day = date.getDate();
    const year = date.getFullYear();
    return `Last modified: ${month} ${day}, ${year}`;
  }

  discardChanges() {
    this.router.navigate(['/strategy']);
  }

  resetStrategy() {
    // Verificar si se puede resetear (plugin no activo)
    if (!this.canSaveStrategy()) {
      this.alertService.showWarning('Cannot reset strategy while plugin is active. Please
deactivate the plugin first.', 'Plugin Active');
      return;
    }

    // Confirmar la acción
    const confirmed = confirm('Are you sure you want to reset the strategy? This will move
all rules back to Available Rules and cannot be undone.');
    if (confirmed) {
      // Resetear a configuración inicial (todas las reglas en Available Rules)
      this.initializeWithConfigAndBalance(initialStrategyState, 0);
    }
  }

  /**
   * MÉTODO 8: Cargar plugin history y verificar si está activo
   * FLUJO DE VERIFICACIÓN:
   * - Cargar plugin history desde Firebase
   * - Verificar si algún plugin está activo
   * - Bloquear botón de guardar si está activo
   */
  setupPluginHistoryListener() {

    if (!this.user?.id) {
```

```
      return;
    }

    try {

      // Suscribirse al Observable del servicio con userId
      this.pluginSubscription =
this.pluginHistoryService.getPluginHistoryRealtime(this.user.id).subscribe({
          next: (pluginHistory: PluginHistory[]) => {
          this.pluginHistory = pluginHistory;

          // Verificar si el plugin está activo usando la nueva lógica de fechas
          if (pluginHistory.length > 0) {
            const plugin = pluginHistory[0];
            this.isPluginActive = this.pluginHistoryService.isPluginActiveByDates(plugin);
          } else {
            // No hay plugin para este usuario
            this.isPluginActive = false;
            console.log('No plugin found for user');
          }

        },
        error: (error) => {
          console.error('Error in plugin history subscription:', error);
          this.isPluginActive = false;
        }
      });

    } catch (error) {
      console.error('Error setting up plugin history listener:', error);
      this.isPluginActive = false;
    }
  }

  /**
   * MÉTODO 9: Verificar si se puede guardar la estrategia
   * FLUJO DE VALIDACIÓN:
   * - Si el plugin está activo, no se puede guardar
   * - Si el plugin no está activo, se puede guardar normalmente
   */
  canSaveStrategy(): boolean {
    const canSave = !this.isPluginActive;
    return canSave;
  }

  /**
   * MÉTODO 10: Limpiar recursos al destruir el componente
   * FLUJO DE LIMPIEZA:
   * - Desuscribirse del listener de Firebase
   * - Evitar memory leaks
   */
  ngOnDestroy() {
    if (this.pluginSubscription) {
      this.pluginSubscription.unsubscribe();
      this.pluginSubscription = null;
    }
  }
}
```

Ø=ÜÁ features\strategy\edit-strategy\components\edit-assets-allowed

Ø=ÜÄ features\strategy\edit-strategy\components\edit-assets-allowed\edit-assets-allowed.component.ts

```ts
import { Component, HostListener, OnInit, Input } from '@angular/core';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../../service/strategy.service';
import {
  assetsAllowed,
  daysAllowed,
  selectMaxDailyTrades,
} from '../../../store/strategy.selectors';
import {
  AssetsAllowedConfig,
  Days,
  DaysAllowedConfig,
  MaxDailyTradesConfig,
  RuleType,
} from '../../../models/strategy.model';
import {
  setAssetsAllowedConfig,
  setDaysAllowedConfig,
  setMaxDailyTradesConfig,
} from '../../../store/strategy.actions';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-edit-assets-allowed',
  templateUrl: './edit-assets-allowed.component.html',
  styleUrls: ['./edit-assets-allowed.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class EditAssetsAllowedComponent implements OnInit {
  @Input() availableSymbolsOptions: string[] = [];

  config: AssetsAllowedConfig = {
    isActive: false,
    type: RuleType.ASSETS_ALLOWED,
    assetsAllowed: ['XMRUSD', 'BTCUSD'],
  };

  symbols: string[] = [];
  selectedInstrument: string | undefined = undefined;
  searchTerm: string = '';

  dropdownOpen = false;

  // Estado de validación
  isValid: boolean = true;
  errorMessage: string = '';

  constructor(
    private store: Store,
    private settingsService: SettingsService
  ) {}

  closeDropdown() {
    this.dropdownOpen = false;
  }

  toggleDropdown() {
    if (this.config.isActive) {
      this.dropdownOpen = !this.dropdownOpen;
    } else {
      this.dropdownOpen = false;
    }
  }
```

```
65
66    onSearchInput(event: Event) {
67      this.searchTerm = (event.target as HTMLInputElement).value;
68      this.dropdownOpen = true;
69    }
70
71    onSearchFocus() {
72      if (this.config.isActive) {
73        this.dropdownOpen = true;
74      }
75    }
76
77    onSearchBlur() {
78      // Delay para permitir click en dropdown
79      setTimeout(() => {
80        this.dropdownOpen = false;
81      }, 200);
82    }
83
84    selectInstrument(intrument: string) {
85      this.selectedInstrument = intrument;
86      this.addSymbol(intrument);
87      this.dropdownOpen = false;
88      this.selectedInstrument = undefined;
89      this.searchTerm = ''; // Limpiar búsqueda
90    }
91
92    getFilteredSymbols(): string[] {
93      if (!this.searchTerm) {
94        return this.availableSymbolsOptions;
95      }
96      return this.availableSymbolsOptions.filter(symbol =>
97        symbol.toLowerCase().includes(this.searchTerm.toLowerCase())
98      );
99    }
100
101   ngOnInit(): void {
102     this.listenRuleConfiguration();
103   }
104
105   addSymbol(symbol: string) {
106     if (symbol && !this.symbols.includes(symbol)) {
107       this.symbols = [...this.symbols, symbol];
108     }
109     this.updateConfig({
110       ...this.config,
111       assetsAllowed: this.symbols,
112     });
113   }
114
115   removeSymbol(symbol: string) {
116     if (this.config.isActive) {
117       this.symbols = this.symbols.filter((s) => s !== symbol);
118       this.updateConfig({
119         ...this.config,
120         assetsAllowed: this.symbols,
121       });
122     }
123   }
124
125   onToggleActive(event: Event) {
126     const isActive = (event.target as HTMLInputElement).checked;
127     const newConfig = {
128       ...this.config,
129       isActive: isActive,
130       // Reiniciar assets cuando se desactiva
131       assetsAllowed: isActive ? this.config.assetsAllowed : [],
132     };
133
134     // Reiniciar símbolos seleccionados
```

```
    if (!isActive) {
      this.symbols = [];
    }

    this.updateConfig(newConfig);
  }

  listenRuleConfiguration() {
    this.store
      .select(assetsAllowed)
      .pipe()
      .subscribe((config) => {
        this.config = { ...config };
        this.symbols = this.config.assetsAllowed;

        // Validar la configuración después de actualizarla
        this.validateConfig(this.config);
      });
  }

  private updateConfig(config: AssetsAllowedConfig) {
    this.store.dispatch(setAssetsAllowedConfig({ config }));
    this.validateConfig(config);
  }

  private validateConfig(config: AssetsAllowedConfig) {

    if (!config.isActive) {
      this.isValid = true;
      this.errorMessage = '';
      return;
    }

    if (!config.assetsAllowed || config.assetsAllowed.length === 0) {
      this.isValid = false;
      this.errorMessage = 'You must select at least one asset';
    } else {
      this.isValid = true;
      this.errorMessage = '';
    }
  }

  // Método público para verificar si la regla es válida
  public isRuleValid(): boolean {
    return this.isValid;
  }

  // Método público para obtener el mensaje de error
  public getErrorMessage(): string {
    return this.errorMessage;
  }

}

```

Ø=ÜÁ features\strategy\edit-strategy\components\edit-days-allowed

Ø=ÜÄ features\strategy\edit-strategy\components\edit-days-allowed\edit-days-allowed.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../../service/strategy.service';
import {
```

```
  daysAllowed,
  selectMaxDailyTrades,
} from '../../../store/strategy.selectors';
import {
  Days,
  DaysAllowedConfig,
  MaxDailyTradesConfig,
  RuleType,
} from '../../../models/strategy.model';
import {
  setDaysAllowedConfig,
  setMaxDailyTradesConfig,
} from '../../../store/strategy.actions';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-edit-days-allowed',
  templateUrl: './edit-days-allowed.component.html',
  styleUrls: ['./edit-days-allowed.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class EditDaysAllowedComponent implements OnInit {
  config: DaysAllowedConfig = {
    isActive: false,
    type: RuleType.DAYS_ALLOWED,
    tradingDays: [],
  };

  daysButtons = [
    { day: Days.MONDAY, isActive: false },
    { day: Days.TUESDAY, isActive: false },
    { day: Days.WEDNESDAY, isActive: false },
    { day: Days.THURSDAY, isActive: false },
    { day: Days.FRIDAY, isActive: false },
    { day: Days.SATURDAY, isActive: false },
    { day: Days.SUNDAY, isActive: false },
  ];

  // Estado de validación
  isValid: boolean = true;
  errorMessage: string = '';

  constructor(private store: Store, private settingsService: SettingsService) {}

  ngOnInit(): void {
    this.listenRuleConfiguration();
  }
  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Reiniciar días cuando se desactiva
      tradingDays: isActive ? this.config.tradingDays : [],
    };

    // Reiniciar botones de días
    if (!isActive) {
      this.daysButtons.forEach(day => {
        day.isActive = false;
      });
    }

    this.updateConfig(newConfig);
  }

  onChangeValue(day: { day: Days; isActive: boolean }) {
    if (this.config.isActive) {
      this.daysButtons.forEach((d) => {
```

```
        if (d.day === day.day) {
          d.isActive = !d.isActive;
        }
      });

      const newConfig: DaysAllowedConfig = {
        ...this.config,
        tradingDays: this.transformDaysActive(),
      };
      this.updateConfig(newConfig);
    }
  }

  transformDaysActive(): string[] {
    let daysArr: string[] = [];

    this.daysButtons.forEach((d) => {
      if (d.isActive) {
        daysArr.push(d.day);
      }
    });

    return daysArr;
  }

  listenRuleConfiguration() {
    this.store
      .select(daysAllowed)
      .pipe()
      .subscribe((config) => {
        this.config = config;
        this.daysButtons.forEach((dayOption) => {
          const findedDay = config.tradingDays.find((d) => d === dayOption.day);
          if (findedDay) {
            dayOption.isActive = true;
          }
        });

        // Validar la configuración después de actualizarla
        this.validateConfig(this.config);
      });
  }

  private updateConfig(config: DaysAllowedConfig) {
    this.store.dispatch(setDaysAllowedConfig({ config }));
    this.validateConfig(config);
  }

  private validateConfig(config: DaysAllowedConfig) {
    if (!config.isActive) {
      this.isValid = true;
      this.errorMessage = '';
      return;
    }

    if (!config.tradingDays || config.tradingDays.length === 0) {
      this.isValid = false;
      this.errorMessage = 'You must select at least one day';
    } else {
      this.isValid = true;
      this.errorMessage = '';
    }
  }

  // Método público para verificar si la regla es válida
  public isRuleValid(): boolean {
    return this.isValid;
  }

  // Método público para obtener el mensaje de error
```

```
  public getErrorMessage(): string {
    return this.errorMessage;
  }
}

```

Ø=ÜÁ features\strategy\edit-strategy\components\edit-hours-allowed

Ø=ÜÄ features\strategy\edit-strategy\components\edit-hours-allowed\edit-hours-allowed.component.ts

```
import { Component, OnInit } from '@angular/core';
import { AlertService } from '../../../../../shared/services/alert.service';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../../service/strategy.service';
import {
  hoursAllowed,
  selectMaxDailyTrades,
} from '../../../store/strategy.selectors';
import {
  HoursAllowedConfig,
  MaxDailyTradesConfig,
  RuleType,
} from '../../../models/strategy.model';
import {
  setHoursAllowedConfig,
  setMaxDailyTradesConfig,
} from '../../../store/strategy.actions';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { NgxMaterialTimepickerModule } from 'ngx-material-timepicker';
import * as moment from 'moment-timezone';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@Component({
  selector: 'app-edit-hours-allowed',
  templateUrl: './edit-hours-allowed.component.html',
  styleUrls: ['./edit-hours-allowed.component.scss'],
  imports: [CommonModule, FormsModule, NgxMaterialTimepickerModule],
  standalone: true,
})
export class EditHoursAllowedComponent implements OnInit {
  config: HoursAllowedConfig = {
    isActive: false,
    tradingOpenTime: '', // Inicialmente vacío para mostrar el placeholder
    tradingCloseTime: '', // Inicialmente vacío para mostrar el placeholder
    timezone: '', // Inicialmente vacío para mostrar el placeholder
    type: RuleType.TRADING_HOURS,
  };

  // Placeholder opcional para el select de timezone
  timezonePlaceholder: string = 'Select a timezone';

  // Estado de validación
  isValid: boolean = true;
  errorMessage: string = '';

  timezones = [
    { value: 'Pacific/Auckland', label: 'Auckland (GMT+12:00)' },
    { value: 'Australia/Sydney', label: 'Sydney (GMT+10:00)' },
    { value: 'Australia/Melbourne', label: 'Melbourne (GMT+10:00)' },
    { value: 'Asia/Tokyo', label: 'Tokyo (GMT+09:00)' },
    { value: 'Asia/Seoul', label: 'Seoul (GMT+09:00)' },
    { value: 'Asia/Shanghai', label: 'Shanghai (GMT+08:00)' },
```

```
    { value: 'Asia/Hong_Kong', label: 'Hong Kong (GMT+08:00)' },
    { value: 'Asia/Singapore', label: 'Singapore (GMT+08:00)' },
    { value: 'Asia/Manila', label: 'Manila (GMT+08:00)' },
    { value: 'Asia/Bangkok', label: 'Bangkok (GMT+07:00)' },
    { value: 'Asia/Jakarta', label: 'Jakarta (GMT+07:00)' },
    { value: 'Asia/Kolkata', label: 'Mumbai (GMT+05:30)' },
    { value: 'Asia/Dubai', label: 'Dubai (GMT+04:00)' },
    { value: 'Europe/Moscow', label: 'Moscow (GMT+03:00)' },
    { value: 'Africa/Cairo', label: 'Cairo (GMT+02:00)' },
    { value: 'Africa/Johannesburg', label: 'Johannesburg (GMT+02:00)' },
    { value: 'Europe/Paris', label: 'Paris (GMT+01:00)' },
    { value: 'Europe/Berlin', label: 'Berlin (GMT+01:00)' },
    { value: 'Europe/Madrid', label: 'Madrid (GMT+01:00)' },
    { value: 'Europe/Rome', label: 'Rome (GMT+01:00)' },
    { value: 'UTC', label: 'UTC (GMT+00:00)' },
    { value: 'Europe/London', label: 'London (GMT+00:00)' },
    { value: 'America/Sao_Paulo', label: 'São Paulo (GMT-03:00)' },
    { value: 'America/New_York', label: 'New York (GMT-05:00)' },
    { value: 'America/Toronto', label: 'Toronto (GMT-05:00)' },
    { value: 'America/Chicago', label: 'Chicago (GMT-06:00)' },
    { value: 'America/Mexico_City', label: 'Mexico City (GMT-06:00)' },
    { value: 'America/Denver', label: 'Denver (GMT-07:00)' },
    { value: 'America/Los_Angeles', label: 'Los Angeles (GMT-08:00)' },
    { value: 'America/Vancouver', label: 'Vancouver (GMT-08:00)' }
  ];

  constructor(private store: Store, private settingsService: SettingsService, private
alertService: AlertService) {}
  ngOnInit(): void {
    this.listenRuleConfiguration();
  }

  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Reiniciar valores cuando se desactiva
      tradingOpenTime: isActive ? this.config.tradingOpenTime : '', // Vacío para mostrar
placeholder
      tradingCloseTime: isActive ? this.config.tradingCloseTime : '', // Vacío para mostrar
placeholder
      timezone: isActive ? this.config.timezone : '', // Vacío para mostrar placeholder
    };
    this.updateConfig(newConfig);
  }
  onTimezoneChange(newTz: string) {
    // Validar que la timezone sea válida
    if (this.isValidTimezone(newTz)) {
      const newConfig = { ...this.config, timezone: newTz };
      this.updateConfig(newConfig);
    } else {
      console.warn('Invalid timezone selected:', newTz);
    }
  }

  isValidTimezone(timezone: string): boolean {
    return this.timezones.some(tz => tz.value === timezone);
  }

  getTimezoneWithCountry(timezone: string): string {
    // Buscar la zona horaria en la lista para obtener el label completo
    const timezoneObj = this.timezones.find(tz => tz.value === timezone);
    return timezoneObj ? timezoneObj.label : timezone;
  }

  onTimeChange(field: 'tradingOpenTime' | 'tradingCloseTime', value: string) {
    const tempConfig = { ...this.config, [field]: value };

    // Solo validar si ambos valores están presentes
    if (tempConfig.tradingOpenTime && tempConfig.tradingCloseTime) {
      const openMinutes = this.toMinutes(tempConfig.tradingOpenTime);
```

```
      const closeMinutes = this.toMinutes(tempConfig.tradingCloseTime);

      if (openMinutes >= closeMinutes) {
        this.alertService.showWarning('Opening time must be earlier than closing time.', 'Invalid Time Range');
        return;
      }
      if (closeMinutes - openMinutes < 30) {
        this.alertService.showWarning('There must be at least a 30-minute difference between opening and closing times.', 'Minimum Time Difference');
        return;
      }
    }

    this.updateConfig(tempConfig);
  }

  listenRuleConfiguration() {
    this.store
      .select(hoursAllowed)
      .pipe()
      .subscribe((config) => {
        // Si la configuración no está activa o es la primera vez, usar valores vacíos para mostrar placeholders
        if (!config.isActive) {
          this.config = {
            ...config,
            tradingOpenTime: '',
            tradingCloseTime: '',
            timezone: ''
          };
        } else {
          this.config = { ...config };
        }

        // Validar la configuración después de actualizarla
        this.validateConfig(this.config);
      });
  }

  private updateConfig(config: HoursAllowedConfig) {
    this.store.dispatch(setHoursAllowedConfig({ config }));
    this.validateConfig(config);
  }

  private validateConfig(config: HoursAllowedConfig) {

    if (!config.isActive) {
      this.isValid = true;
      this.errorMessage = '';
      return;
    }

    const missingFields = [];

    if (!config.tradingOpenTime || config.tradingOpenTime.trim() === '') {
      missingFields.push('start time');
    }

    if (!config.tradingCloseTime || config.tradingCloseTime.trim() === '') {
      missingFields.push('end time');
    }

    if (!config.timezone || config.timezone.trim() === '') {
      missingFields.push('timezone');
    }

    if (missingFields.length > 0) {
      this.isValid = false;
      this.errorMessage = `Please fill in the following fields: ${missingFields.join(', ')}`;
    } else {
      this.isValid = true;
      this.errorMessage = '';
```

```
    }
  }

  private toMinutes(time: string): number {
    if (!time || time.trim() === '') {
      return 0;
    }

    const is12hFormat =
      time.toUpperCase().includes('AM') || time.toUpperCase().includes('PM');

    let hours: number;
    let minutes: number;

    if (is12hFormat) {
      const [timePart, period] = time.split(' ');
      [hours, minutes] = timePart.split(':').map(Number);
      if (period.toUpperCase() === 'PM' && hours !== 12) {
        hours += 12;
      }
      if (period.toUpperCase() === 'AM' && hours === 12) {
        hours = 0;
      }
    } else {
      [hours, minutes] = time.split(':').map(Number);
    }

    return hours * 60 + minutes;
  }

  // Método público para verificar si la regla es válida
  public isRuleValid(): boolean {
    return this.isValid;
  }

  // Método público para obtener el mensaje de error
  public getErrorMessage(): string {
    return this.errorMessage;
  }
}
```

Ø=ÜÁ features\strategy\edit-strategy\components\edit-max-daily-trades

Ø=ÜÄ features\strategy\edit-strategy\components\edit-max-daily-trades\edit-max-daily-trades.component.ts

```ts
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../../service/strategy.service';
import { selectMaxDailyTrades } from '../../../store/strategy.selectors';
import { MaxDailyTradesConfig, RuleType } from '../../../models/strategy.model';
import { setMaxDailyTradesConfig } from '../../../store/strategy.actions';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-edit-max-daily-trades',
  templateUrl: './edit-max-daily-trades.component.html',
  styleUrls: ['./edit-max-daily-trades.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class EditMaxDailyTradesComponent implements OnInit {
  config: MaxDailyTradesConfig = {
```

```
    isActive: false,
    maxDailyTrades: 0, // Cambiar el valor por defecto a 0
    type: RuleType.MAX_DAILY_TRADES,
  };

  // Estado de validación
  isValid: boolean = true;
  errorMessage: string = '';

  constructor(private store: Store, private settingsService: SettingsService) {}

  ngOnInit(): void {
    this.listenRuleConfiguration();
  }

  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Resetear a 0 cuando se desactiva
      maxDailyTrades: isActive ? this.config.maxDailyTrades : 0,
    };
    this.updateConfig(newConfig);
  }

  onChangeValue(event: Event) {
    const numValue = Number((event.target as HTMLInputElement).value);
    const newConfig: MaxDailyTradesConfig = {
      ...this.config,
      maxDailyTrades: numValue < 1 ? 1 : numValue,
    };
    this.updateConfig(newConfig);
  }

  // Métodos para spinner (solo incrementar/decrementar)
  incrementValue() {
    if (this.config.isActive) {
      const newConfig: MaxDailyTradesConfig = {
        ...this.config,
        maxDailyTrades: this.config.maxDailyTrades + 1,
      };
      this.updateConfig(newConfig);
    }
  }

  decrementValue() {
    if (this.config.isActive && this.config.maxDailyTrades > 0) {
      const newConfig: MaxDailyTradesConfig = {
        ...this.config,
        maxDailyTrades: this.config.maxDailyTrades - 1,
      };
      this.updateConfig(newConfig);
    }
  }

  listenRuleConfiguration() {
    this.store
      .select(selectMaxDailyTrades)
      .pipe()
      .subscribe((config) => {
        // Si la configuración no está activa, usar valor 0 para mostrar placeholder
        if (!config.isActive) {
          this.config = {
            ...config,
            maxDailyTrades: 0
          };
        } else {
          this.config = config;
        }
```

```
88
89          // Validar la configuración después de actualizarla
90          this.validateConfig(this.config);
91        });
92    }
93
94    private updateConfig(config: MaxDailyTradesConfig) {
95      this.store.dispatch(setMaxDailyTradesConfig({ config }));
96      this.validateConfig(config);
97    }
98
99    private validateConfig(config: MaxDailyTradesConfig) {
100     if (!config.isActive) {
101       this.isValid = true;
102       this.errorMessage = '';
103       return;
104     }
105
106     if (config.maxDailyTrades <= 0) {
107       this.isValid = false;
108       this.errorMessage = 'You must have at least one trade per day';
109     } else {
110       this.isValid = true;
111       this.errorMessage = '';
112     }
113   }
114
115   // Método público para verificar si la regla es válida
116   public isRuleValid(): boolean {
117     return this.isValid;
118   }
119
120   // Método público para obtener el mensaje de error
121   public getErrorMessage(): string {
122     return this.errorMessage;
123   }
124 }
125
```

Ø=ÜÁ features\strategy\edit-strategy\components\edit-risk-per-trade

Ø=ÜÄ features\strategy\edit-strategy\components\edit-risk-per-trade\edit-risk-per-trade.component.ts

```
1  import { CommonModule } from '@angular/common';
2  import { Component, HostListener, OnInit, ViewChild, ElementRef, Input } from '@angular/
   core';
3  import { take } from 'rxjs/operators';
4  import {
5    MaxDailyTradesConfig,
6    RiskPerTradeConfig,
7    RiskRewardConfig,
8    RuleType,
9  } from '../../../models/strategy.model';
10 import { Store } from '@ngrx/store';
11 import { SettingsService } from '../../../service/strategy.service';
12 import { riskPerTrade, riskReward } from '../../../store/strategy.selectors';
13 import {
14   setRiskPerTradeConfig,
15   setRiskRewardConfig,
16 } from '../../../store/strategy.actions';
17 import { currencies } from './models/risk-per-trade.model';
18 import { ReportService } from '../../../../report/service/report.service';
19 import { AppContextService } from '../../../../../shared/context/context';
20 import { AuthService } from '../../../../auth/service/authService';
```

```
import { AccountData } from '../../../../auth/models/userModel';
import { NumberFormatterService } from '../../../../../shared/utils/number-
formatter.service';
@Component({
  selector: 'app-edit-risk-per-trade',
  templateUrl: './edit-risk-per-trade.component.html',
  styleUrls: ['./edit-risk-per-trade.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
export class EditRiskPerTradeComponent implements OnInit {
  @Input() userAccounts: AccountData[] = [];

  config: RiskPerTradeConfig = {
    isActive: false,
    review_type: 'MAX',
    number_type: 'PERCENTAGE',
    percentage_type: 'NULL',
    risk_ammount: 0,
    type: RuleType.MAX_RISK_PER_TRADE,
    balance: 0,
    actualBalance: 0,
  };

  // Nuevas propiedades para la lógica del componente
  selectedSizeType: 'max-size' | 'fixed' | null = null;
  selectedCalculationType: 'by percentage' | 'by price' | null = null;
  selectedBalanceType: 'by actual balance' | 'by initial balance' | null = null;

  // Valores del balance actual (se obtiene del servicio)
  actualBalance: number = 0;
  initialBalance: number = 0;
  // Balances por cuenta
  accountActualBalances: Record<string, number> = {};
  selectedAccount: AccountData | null = null;

  // Variables para el balance inicial
  initialBalanceValue: number = 0;
  displayInitialBalanceValue: string = '';
  isInitialBalanceEditing: boolean = true; // Iniciar en modo edición
  isInitialBalanceConfirmed: boolean = false;

  // Valores de entrada
  percentageValue: number = 0;
  priceValue: number = 0;

  // Valor mostrado en el input de precio (con formato)
  displayPriceValue: string = '';

  // Valor del input de precio sin formato (para edición)
  priceInputValue: string = '';

  // Propiedad para el precio formateado
  price: string = '';

  inputFirstRatioValue: number = 0;
  inputSecondRatioValue: number = 0;
  initialRiskTrade: number | undefined;
  selectedCurrency = currencies[0];
  dropdownOpen = false;
  currencies = currencies;

  // Estado de validación
  isValid: boolean = true;
  errorMessage: string = '';

  // Rastrear el estado inicial de Firebase
  private initialFirebaseState: boolean | null = null;

  // Control de peticiones para evitar llamadas infinitas
```

```
  private accountLoadAttempts: number = 0;
  private maxAccountLoadAttempts: number = 2;

  // Referencia al dropdown de cuenta
  @ViewChild('accountSelect') accountSelect?: ElementRef<HTMLSelectElement>;

  constructor(
    private store: Store,
    private settingsService: SettingsService,
    private reportService: ReportService,
    private appContext: AppContextService,
    private authService: AuthService,
    private numberFormatter: NumberFormatterService
  ) {}

  closeDropdown() {
    this.dropdownOpen = false;
  }

  ngOnInit(): void {
    // Capturar el estado inicial ANTES de suscribirse al observable
    this.captureInitialFirebaseState();
    this.listenRuleConfiguration();

    // Inicializar cuentas si vienen como input
    this.initializeUserAccounts();
  }

  /**
   * Inicializar cuentas del usuario desde el input
   */
  private initializeUserAccounts(): void {
    if (this.userAccounts && this.userAccounts.length > 0) {
      // Las cuentas ya vienen cargadas desde el componente padre
      console.log('User accounts loaded from parent:', this.userAccounts);
    } else {
      // Si no hay cuentas, cargar desde el servicio (fallback)
      this.loadUserAccounts();
    }
  }


  /**
   * Capturar el estado inicial de Firebase desde el store actual
   * Esto evita que se capture múltiples veces durante las emisiones del observable
   */
  private captureInitialFirebaseState(): void {
    // Usar take(1) para obtener solo el primer valor y luego desuscribirse automáticamente
    this.store.select(riskPerTrade).pipe(take(1)).subscribe(config => {
      this.initialFirebaseState = config.isActive;
    });
  }

  toggleDropdown() {
    if (this.config.isActive) {
      this.dropdownOpen = !this.dropdownOpen;
    } else {
      this.dropdownOpen = false;
    }
  }

  selectCurrency(currency: { code: string; country: string }) {
    this.selectedCurrency = currency;
    this.dropdownOpen = false;
  }

  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;

    if (!isActive) {
```

```
      // Si se desactiva, resetear todos los valores a 0
      const newConfig = {
        ...this.config,
        isActive: false,
        balance: 0,
        actualBalance: 0,
        risk_ammount: 0,
        review_type: 'MAX' as const,
        number_type: 'PERCENTAGE' as const,
        percentage_type: 'NULL' as const,
      };
      this.updateConfig(newConfig);
    } else {
      // Si se activa, mantener la configuración actual
      const newConfig = {
        ...this.config,
        isActive: true,
      };
      this.updateConfig(newConfig);
    }
  }

  // Nuevos métodos para manejar las opciones
  selectSizeType(type: 'max-size' | 'fixed') {
    this.selectedSizeType = type;
    this.selectedCalculationType = null;
    this.selectedBalanceType = null;
    this.saveConfiguration();
  }

  selectCalculationType(type: 'by percentage' | 'by price') {
    this.selectedCalculationType = type;

    // Resetear todos los campos relacionados cuando se cambia el tipo de cálculo
    this.selectedBalanceType = null;
    this.selectedAccount = null;
    this.percentageValue = 0;
    this.priceValue = 0;
    this.displayPriceValue = '';
    this.priceInputValue = '';
    this.initialBalance = 0;
    this.initialBalanceValue = 0;
    this.displayInitialBalanceValue = '';
    this.isInitialBalanceConfirmed = false;
    this.isInitialBalanceEditing = true;

    // Resetear el dropdown para mostrar el placeholder
    setTimeout(() => {
      if (this.accountSelect) {
        this.accountSelect.nativeElement.selectedIndex = 0; // Seleccionar la primera opción
(placeholder)
      }
    }, 0);

    this.saveConfiguration();
  }

  selectBalanceType(type: 'by actual balance' | 'by initial balance') {
    this.selectedBalanceType = type;

    // Resetear cuenta seleccionada y valores cuando se cambia el tipo de balance
    this.selectedAccount = null;
    this.percentageValue = 0;
    this.priceValue = 0;
    this.displayPriceValue = '';
    this.priceInputValue = '';
    this.initialBalance = 0;
    this.initialBalanceValue = 0;
    this.displayInitialBalanceValue = '';
    this.isInitialBalanceConfirmed = false;
    this.isInitialBalanceEditing = true;
```

```
231
232     // Resetear el dropdown para mostrar el placeholder
233     setTimeout(() => {
234       if (this.accountSelect) {
235         this.accountSelect.nativeElement.selectedIndex = 0; // Seleccionar la primera opción
236 (placeholder)
237       }
238     }, 0);
239
240     // Solo cargar el balance actual cuando el usuario seleccione "by actual balance"
241     if (type === 'by actual balance') {
242       if (this.userAccounts.length > 0) {
243         this.loadActualBalancesForAccounts();
244       } else {
245         this.loadUserAccounts().then(() => this.loadActualBalancesForAccounts());
246       }
247     } else {
248       // by initial balance -> las cuentas ya están disponibles desde el input
249       if (this.userAccounts.length === 0) {
250         this.loadUserAccounts();
251       }
252     }
253
254     this.saveConfiguration();
255   }
256
257   async loadUserAccounts() {
258     try {
259       // Preferir UID de Firebase
260       const firebaseUser = this.authService.getAuth().currentUser;
261       const userId = firebaseUser?.uid || this.appContext.currentUser()?.id;
262       if (!userId) {
263         this.userAccounts = [];
264         return;
265       }
266       const accounts = await this.authService.getUserAccounts(userId);
267       this.userAccounts = accounts || [];
268     } catch {
269       this.userAccounts = [];
270     }
271   }
272
273   async loadActualBalancesForAccounts() {
274     try {
275       const balances: Record<string, number> = {};
276       for (const acc of this.userAccounts) {
277         try {
278           // Obtener token de TradeLocker por cuenta (igual que en ReportService)
279           const token = await this.reportService
280             .getUserKey(acc.emailTradingAccount, acc.brokerPassword, acc.server)
281             .toPromise();
282           if (!token) {
283             balances[acc.accountID] = 0;
284             continue;
285           }
286           // Con ese token, obtener el balance de la cuenta
287           const data = await this.reportService
288             .getBalanceData(acc.accountID, token, acc.accountNumber)
289             .toPromise();
290           balances[acc.accountID] = data?.balance || 0;
291         } catch {
292           balances[acc.accountID] = 0;
293         }
294       }
295       this.accountActualBalances = balances;
296     } catch {
297       this.accountActualBalances = {};
298     }
299   }
300
   onSelectAccount(event: Event) {
```

```
    const accountID = (event.target as HTMLSelectElement).value;
    this.selectedAccount = this.userAccounts.find(a => a.accountID === accountID) || null;

    // Asegurar que se tengan los tipos necesarios para mostrar el input de porcentaje
    if (!this.selectedSizeType) {
      this.selectedSizeType = 'max-size'; // Valor por defecto
    }
    if (!this.selectedCalculationType) {
      this.selectedCalculationType = 'by percentage'; // Valor por defecto
    }

    // Ajustar balances visibles según selección
    if (this.selectedBalanceType === 'by initial balance') {
      this.initialBalance = this.selectedAccount?.initialBalance || 0;
      // Mostrar inmediatamente el balance inicial seleccionado en los resúmenes
      this.isInitialBalanceConfirmed = true;
      this.isInitialBalanceEditing = false;
    } else if (this.selectedBalanceType === 'by actual balance' && this.selectedAccount) {
      this.actualBalance = this.accountActualBalances[this.selectedAccount.accountID] || 0;
    }
    this.saveConfiguration();
  }

  onPercentageChange(event: Event) {
    this.percentageValue = Number((event.target as HTMLInputElement).value);
    this.saveConfiguration();
  }

  formatCurrency(event: any) {
    const input = event.target.value;
    const formatted = this.numberFormatter.formatInputValue(input);

    if (formatted === '') {
      this.price = '';
      this.priceValue = 0;
      this.displayPriceValue = '';
      return;
    }

    // Convertir a número
    const value = parseFloat(this.numberFormatter.cleanNumericInput(input));
    if (isNaN(value)) {
      this.price = '';
      this.priceValue = 0;
      this.displayPriceValue = '';
      return;
    }

    this.priceValue = value;
    this.price = formatted;
    this.displayPriceValue = this.price;
    this.saveConfiguration();
  }

  onPriceInput(event: Event) {
    const target = event.target as HTMLInputElement;
    this.priceInputValue = target.value;
  }

  onPriceFocus() {
    // Cuando el usuario hace focus, mostrar solo el número sin formato para edición
    if (this.priceValue > 0) {
      this.priceInputValue = this.priceValue.toString();
    }
  }

  onPriceBlur() {
    // Convertir el valor a número usando el servicio centralizado
    const numericValue = this.numberFormatter.parseCurrencyValue(this.priceInputValue);
    if (!isNaN(numericValue) && numericValue > 0) {
```

```
      // Guardar el valor sin formato
      this.priceValue = numericValue;

      // Mostrar formato visual (solo para display)
      this.displayPriceValue = this.numberFormatter.formatCurrencyDisplay(numericValue);

      // Actualizar el input para mostrar el formato visual
      this.priceInputValue = this.displayPriceValue;

      this.saveConfiguration();
    } else {
      // Si no es un número válido, limpiar
      this.priceInputValue = '';
      this.displayPriceValue = '';
      this.priceValue = 0;
    }
  }

  onBlur() {
    // Método legacy - mantener para compatibilidad
    this.onPriceBlur();
  }

  // Métodos para el balance inicial
  formatInitialBalance(event: any) {
    const input = event.target.value;
    const formatted = this.numberFormatter.formatInputValue(input);

    if (formatted === '') {
      this.initialBalanceValue = 0;
      this.displayInitialBalanceValue = '';
      return;
    }

    // Convertir a número
    const value = parseFloat(this.numberFormatter.cleanNumericInput(input));
    if (isNaN(value)) {
      this.initialBalanceValue = 0;
      this.displayInitialBalanceValue = '';
      return;
    }

    this.initialBalanceValue = value;
    this.displayInitialBalanceValue = formatted;
  }

  onInitialBalanceBlur() {
    if (!this.displayInitialBalanceValue) return;

    const num = this.numberFormatter.parseCurrencyValue(this.displayInitialBalanceValue);
    if (!isNaN(num)) {
      this.initialBalanceValue = num;
      this.displayInitialBalanceValue = this.numberFormatter.formatNumber(num, 2);
    }
  }

  clearInitialBalance() {
    this.initialBalanceValue = 0;
    this.displayInitialBalanceValue = '';
    this.initialBalance = 0;
    this.isInitialBalanceConfirmed = false;
    this.isInitialBalanceEditing = true; // Habilitar edición después de borrar
  }

  editInitialBalance() {
    this.isInitialBalanceEditing = true;
    this.isInitialBalanceConfirmed = false;
  }

  confirmInitialBalance() {
```

```
    this.initialBalance = this.initialBalanceValue;
    this.isInitialBalanceEditing = false;
    this.isInitialBalanceConfirmed = true;
    this.saveConfiguration();
  }

  // Método legacy para cargar un balance actual (se mantiene por compatibilidad, ahora usamos loadActualBalancesForAccounts)
  async loadActualBalance() {
    try {
      // Obtener el usuario actual de Firebase directamente
      const currentUser = this.authService.getAuth().currentUser;
      if (!currentUser) {
        this.actualBalance = 0;
        return;
      }

      // Obtener el token fresco
      const accessToken = await currentUser.getIdToken(true); // true = force refresh

      // Obtener las cuentas del usuario desde el contexto
      const userAccounts = this.appContext.userAccounts();
      if (!userAccounts || userAccounts.length === 0) {
        this.actualBalance = 0;
        return;
      }

      const account = userAccounts[0];

      // Hacer la petición al servicio de reportes para obtener el balance
      const balanceData = await this.reportService.getBalanceData(
        account.accountID,
        accessToken,
        account.accountNumber
      ).toPromise();

      if (balanceData && balanceData.balance) {
        this.actualBalance = balanceData.balance;
        // Actualizar el contexto con los datos obtenidos
        this.appContext.updateReportBalance(balanceData);
      } else {
        this.actualBalance = 0;
      }
    } catch (error) {
      console.error('Error loading actual balance:', error);
      this.actualBalance = 0;
    }
  }

  getCurrentBalance(): number {
    if (this.selectedBalanceType === 'by actual balance') {
      return this.actualBalance;
    } else if (this.selectedBalanceType === 'by initial balance') {
      return this.isInitialBalanceConfirmed ? this.initialBalance : 0;
    }
    return 0;
  }

  getCalculatedAmount(): number {
    const balance = this.getCurrentBalance();
    return (this.percentageValue / 100) * balance;
  }

  onChangePercentage(event: Event) {
    const percentage = Number((event.target as HTMLInputElement).value);
    const moneyRisk = Number(
      ((percentage / 100) * this.getCurrentBalance()).toFixed(2)
    );

    // Usar la misma lógica que saveConfiguration para determinar el balance
    let balanceToSave: number;
```

```
    let actualBalanceToSave: number | undefined;
    if (this.selectedCalculationType === 'by percentage' && this.selectedBalanceType === 'by initial balance') {
      balanceToSave = this.selectedAccount?.initialBalance || 0;
      actualBalanceToSave = 0;
    } else {
      balanceToSave = -1;
      actualBalanceToSave = this.actualBalance;
    }

    const newConfig: RiskPerTradeConfig = {
      ...this.config,
      risk_ammount: percentage,
      balance: balanceToSave,
      actualBalance: actualBalanceToSave,
    };
    this.updateConfig(newConfig);
  }

  onChangeAmount(event: Event) {
    const moneyRisk = Number((event.target as HTMLInputElement).value);
    const percentage = Number(
      ((moneyRisk / this.getCurrentBalance()) * 100).toFixed(2)
    );

    // Cuando es money, siempre guardar 0 en ambos
    const newConfig: RiskPerTradeConfig = {
      ...this.config,
      risk_ammount: moneyRisk,
      balance: 0,
      actualBalance: 0,
    };
    this.updateConfig(newConfig);
  }

  listenRuleConfiguration() {
    this.store
      .select(riskPerTrade)
      .pipe()
      .subscribe((config) => {
        this.config = config;

        // Resetear contador de intentos cuando se carga una nueva configuración
        this.accountLoadAttempts = 0;

        // El estado inicial ya fue capturado en ngOnInit, solo procesar la lógica

        if (config.isActive) {
          // Usar el estado inicial de Firebase para determinar el comportamiento
          if (this.initialFirebaseState === false) {
            // Regla vino inactiva de Firebase: SIEMPRE empezar desde cero, ignorar valores de config
            this.selectedSizeType = null;
            this.selectedCalculationType = null;
            this.selectedBalanceType = null;
            this.percentageValue = 0;
            this.priceValue = 0;
            this.displayPriceValue = '';
            this.priceInputValue = '';
            this.initialBalance = 0;
            this.initialBalanceValue = 0;
            this.displayInitialBalanceValue = '';
            this.isInitialBalanceConfirmed = false;
            this.isInitialBalanceEditing = true;
            this.selectedAccount = null;
            this.actualBalance = 0;
          } else {
            // Regla vino activa de Firebase: cargar valores desde config
            this.selectedSizeType = config.review_type === 'MAX' ? 'max-size' : 'fixed';
            this.selectedCalculationType = config.number_type === 'PERCENTAGE' ? 'by percentage' : 'by price';
            // Manejar el percentage_type correctamente
```

```
            if (config.percentage_type === 'NULL') {
              // Cuando es money, no hay balance type específico
              this.selectedBalanceType = null;
            } else {
              this.selectedBalanceType = config.percentage_type === 'ACTUAL_B' ? 'by actual balance' : 'by initial balance';
            }

            // Cargar el valor de riesgo
            if (config.risk_ammount) {
              if (config.number_type === 'PERCENTAGE') {
                this.percentageValue = config.risk_ammount;
              } else {
                this.priceValue = config.risk_ammount;
                this.price = config.risk_ammount.toLocaleString('en-US', {
                  style: 'currency',
                  currency: 'USD',
                  minimumFractionDigits: 2,
                  maximumFractionDigits: 2
                });
                this.displayPriceValue = this.price;
                this.priceInputValue = this.displayPriceValue;
              }
            }

            // Cargar balance y seleccionar cuenta según el tipo
            if (config.percentage_type === 'INITIAL_B' && config.balance && config.balance > 0) {
              // Configurar el tipo de balance y cálculo para balance inicial
              this.selectedBalanceType = 'by initial balance';
              this.selectedCalculationType = 'by percentage';

              // Cargar balance inicial y seleccionar la cuenta correspondiente
              this.initialBalance = config.balance;
              this.initialBalanceValue = config.balance;

              this.displayInitialBalanceValue = config.balance.toLocaleString('en-US', {
                minimumFractionDigits: 2,
                maximumFractionDigits: 2
              });
              this.isInitialBalanceConfirmed = true;
              this.isInitialBalanceEditing = false;

              // Usar cuentas disponibles o cargarlas si es necesario
              if (this.userAccounts.length > 0) {

                // Buscar la cuenta que coincida con el balance inicial
                const matchingAccount = this.userAccounts.find((account) => {
                  // Asegurar que ambos valores sean números
                  const accountBalance = Number(account.initialBalance) || 0;
                  const configBalance = Number(config.balance) || 0;
                  const difference = Math.abs(accountBalance - configBalance);
                  return difference < 0.01; // Tolerancia para decimales
                });

                if (matchingAccount) {
                  this.selectedAccount = matchingAccount;
                  // Seleccionar en el dropdown
                  setTimeout(() => {
                    if (this.accountSelect) {
                      const selectElement = this.accountSelect.nativeElement;
                      const optionIndex = this.userAccounts.findIndex(account =>
                        account.accountID === matchingAccount.accountID
                      );
                      if (optionIndex !== -1) {
                        selectElement.selectedIndex = optionIndex + 1;
                      }
                    }
                  }, 100);
                } else {
                  this.selectedAccount = null;
                  this.isValid = false;
```

```
                this.errorMessage = 'Please complete this rule. The saved balance does not match any available account.';
              }
            } else {
              // Controlar intentos de carga para evitar peticiones infinitas
              if (this.accountLoadAttempts < this.maxAccountLoadAttempts) {
                this.accountLoadAttempts++;
                this.loadUserAccounts().then(() => {
                  // Buscar la cuenta que coincida con el balance inicial
                  const matchingAccount = this.userAccounts.find((account) => {
                    // Asegurar que ambos valores sean números
                    const accountBalance = Number(account.initialBalance) || 0;
                    const configBalance = Number(config.balance) || 0;
                    const difference = Math.abs(accountBalance - configBalance);
                    return difference < 0.01; // Tolerancia para decimales
                  });

                  if (matchingAccount) {
                    this.selectedAccount = matchingAccount;
                    // Seleccionar en el dropdown
                    setTimeout(() => {
                      if (this.accountSelect) {
                        const selectElement = this.accountSelect.nativeElement;
                        const optionIndex = this.userAccounts.findIndex(account =>
                          account.accountID === matchingAccount.accountID
                        );
                        if (optionIndex !== -1) {
                          selectElement.selectedIndex = optionIndex + 1; // +1 porque el primer option es el placeholder
                        }
                      }
                    }, 100);
                  } else {
                    this.selectedAccount = null;
                    this.isValid = false;
                    this.errorMessage = 'Please complete this rule. The saved balance does not match any available account.';
                  }
                }).catch(() => {
                  // Si falla la carga, mostrar error
                  this.selectedAccount = null;
                  this.isValid = false;
                  this.errorMessage = 'Error loading accounts. Please try again.';
                });
              } else {
                // Máximo de intentos alcanzado, mostrar error
                this.selectedAccount = null;
                this.isValid = false;
                this.errorMessage = 'Please complete this rule. The saved balance does not match any available account.';
              }
            }

          } else if (config.percentage_type === 'ACTUAL_B' && config.actualBalance.balance && config.actualBalance.balance > 0) {
            // Configurar el tipo de balance y cálculo para balance actual
            this.selectedBalanceType = 'by actual balance';
            this.selectedCalculationType = 'by percentage';

            // Cargar balance actual y seleccionar la cuenta correspondiente
            this.actualBalance = config.actualBalance.balance || 0;

            // Usar cuentas disponibles o cargarlas si es necesario
            if (this.userAccounts.length > 0) {
              this.loadActualBalancesForAccounts().then(() => {
                // Buscar la cuenta que tenga el balance actual que coincida
                const matchingAccount = this.userAccounts.find(account => {
                  // Asegurar que ambos valores sean números
                  const accountBalance = this.accountActualBalances[account.accountID] || 0;
                  const configBalance = Number(config.actualBalance.balance) || 0;
                  const difference = Math.abs(accountBalance - configBalance);
                  return difference < 0.01; // Tolerancia para decimales
                });

                if (matchingAccount) {
```

```
                    this.selectedAccount = matchingAccount;
                    // Seleccionar en el dropdown
                    setTimeout(() => {
                      if (this.accountSelect) {
                        const selectElement = this.accountSelect.nativeElement;
                        const optionIndex = this.userAccounts.findIndex(account =>
                          account.accountID === matchingAccount.accountID
                        );
                        if (optionIndex !== -1) {
                          selectElement.selectedIndex = optionIndex + 1; // +1 porque el primer option es el placeholder
                        }
                      }
                    }, 100);
                  } else {
                    this.selectedAccount = null;
                    this.isValid = false;
                    this.errorMessage = 'Please complete this rule. The saved balance does not match any available account.';
                  }
                });
              } else {
                // Controlar intentos de carga para evitar peticiones infinitas
                if (this.accountLoadAttempts < this.maxAccountLoadAttempts) {
                  this.accountLoadAttempts++;
                  this.loadUserAccounts().then(() => {
                    this.loadActualBalancesForAccounts().then(() => {
                      // Buscar la cuenta que tenga el balance actual que coincida
                      const matchingAccount = this.userAccounts.find(account => {
                        // Asegurar que ambos valores sean números
                        const accountBalance = Number(this.accountActualBalances[account.accountID]) || 0;
                        const configBalance = Number(config.actualBalance.balance) || 0;
                        const difference = Math.abs(accountBalance - configBalance);
                        return difference < 0.01; // Tolerancia para decimales
                      });

                      if (matchingAccount) {
                        this.selectedAccount = matchingAccount;
                        // Seleccionar en el dropdown
                        setTimeout(() => {
                          if (this.accountSelect) {
                            const selectElement = this.accountSelect.nativeElement;
                            const optionIndex = this.userAccounts.findIndex(account =>
                              account.accountID === matchingAccount.accountID
                            );
                            if (optionIndex !== -1) {
                              selectElement.selectedIndex = optionIndex + 1; // +1 porque el primer option es el placeholder
                            }
                          }
                        }, 100);
                      } else {
                        this.selectedAccount = null;
                        this.isValid = false;
                        this.errorMessage = 'Please complete this rule. The saved balance does not match any available account.';
                      }
                    }).catch(() => {
                      // Si falla la carga de balances, mostrar error
                      this.selectedAccount = null;
                      this.isValid = false;
                      this.errorMessage = 'Error loading account balances. Please try again.';
                    });
                  }).catch(() => {
                    // Si falla la carga de cuentas, mostrar error
                    this.selectedAccount = null;
                    this.isValid = false;
                    this.errorMessage = 'Error loading accounts. Please try again.';
                  });
                } else {
                  // Máximo de intentos alcanzado, mostrar error
                  this.selectedAccount = null;
                  this.isValid = false;
                  this.errorMessage = 'Please complete this rule. The saved balance does not match any available account.';
```

```
                }
              }
            }
          }

        } else {
          // Si la regla está desactivada, resetear UI a null
          this.selectedSizeType = null;
          this.selectedCalculationType = null;
          this.selectedBalanceType = null;
          this.percentageValue = 0;
          this.priceValue = 0;
          this.displayPriceValue = '';
          this.priceInputValue = '';
          this.initialBalance = 0;
          this.initialBalanceValue = 0;
          this.displayInitialBalanceValue = '';
          this.isInitialBalanceConfirmed = false;
          this.isInitialBalanceEditing = true;
          this.selectedAccount = null;
          this.actualBalance = 0;
        }

        // Validar la configuración después de actualizarla (solo si no hay selección automática pendiente)
        // La validación para casos con selección automática se maneja dentro de los setTimeout
        if (!config.isActive ||
            (config.percentage_type !== 'INITIAL_B' && config.percentage_type !== 'ACTUAL_B')) {
          this.validateConfig(this.config);
        }
      });
  }

  private updateConfig(config: RiskPerTradeConfig) {
    this.store.dispatch(setRiskPerTradeConfig({ config }));
    this.validateConfig(config);
  }

  private validateConfig(config: RiskPerTradeConfig) {
    if (!config.isActive) {
      this.isValid = true;
      this.errorMessage = '';
      return;
    }

    // Validar que se haya seleccionado un tipo de tamaño
    if (!this.selectedSizeType) {
      this.isValid = false;
      this.errorMessage = 'You must select a size type';
      return;
    }

    // Validar que se haya seleccionado un tipo de cálculo
    if (!this.selectedCalculationType) {
      this.isValid = false;
      this.errorMessage = 'You must select a calculation type';
      return;
    }

    // Validar según el tipo de cálculo seleccionado
    if (this.selectedCalculationType === 'by percentage') {
      // Validar que se haya seleccionado un tipo de balance
      if (!this.selectedBalanceType) {
        this.isValid = false;
        this.errorMessage = 'You must select a balance type';
        return;
      }

      // Validar que se haya seleccionado una cuenta
      if (!this.selectedAccount) {
        this.isValid = false;
```

```
        this.errorMessage = 'You must select an account';
        return;
      }

      // Validar que se haya ingresado un porcentaje válido
      if (!this.percentageValue || this.percentageValue <= 0) {
        this.isValid = false;
        this.errorMessage = 'You must enter a valid percentage value';
        return;
      }
    } else if (this.selectedCalculationType === 'by price') {
      // Validar que se haya ingresado un precio válido
      if (!this.priceValue || this.priceValue <= 0) {
        this.isValid = false;
        this.errorMessage = 'You must enter a valid price value';
        return;
      }
    }

    // Si llegamos aquí, la validación pasó
    this.isValid = true;
    this.errorMessage = '';
  }

  // Método público para verificar si la regla es válida
  public isRuleValid(): boolean {
    return this.isValid;
  }

  // Método público para obtener el mensaje de error
  public getErrorMessage(): string {
    return this.errorMessage;
  }

  // Método para guardar la configuración con la nueva estructura
  private saveConfiguration() {
    if (!this.selectedSizeType || !this.selectedCalculationType) {
      return; // No guardar si faltan selecciones básicas
    }

    // Para percentage, también necesitamos selectedBalanceType
    if (this.selectedCalculationType === 'by percentage' && !this.selectedBalanceType) {
      return;
    }

    // Determinar el balance a guardar según la lógica especificada
    let balanceToSave: number;
    let actualBalanceToSave: number | undefined;
    let percentageType: "INITIAL_B" | "ACTUAL_B" | "NULL";

    if (this.selectedCalculationType === 'by percentage') {
      // Cuando es percentage
      if (this.selectedBalanceType === 'by initial balance') {
        // percentage + initialBalance: guardar el balance en balance, actualBalance = 0
        balanceToSave = this.selectedAccount?.initialBalance || 0;
        actualBalanceToSave = 0;
        percentageType = 'INITIAL_B';
      } else {
        // percentage + actualBalance: guardar -1 en balance, valor en actualBalance
        balanceToSave = -1;
        actualBalanceToSave = this.actualBalance;
        percentageType = 'ACTUAL_B';
      }
    } else {
      // Cuando es money: guardar 0 en ambos
      balanceToSave = 0;
      actualBalanceToSave = 0;
      percentageType = 'NULL';
    }

```

```
    const newConfig: RiskPerTradeConfig = {
      ...this.config,
      review_type: this.selectedSizeType === 'max-size' ? 'MAX' : 'FIXED',
      number_type: this.selectedCalculationType === 'by percentage' ? 'PERCENTAGE' : 'MONEY',
      percentage_type: percentageType,
      risk_ammount: this.selectedCalculationType === 'by percentage' ?
this.percentageValue : this.priceValue,
      balance: balanceToSave,
      actualBalance: actualBalanceToSave
    };

    this.updateConfig(newConfig);
  }
}
```

Ø=ÜÁ features\strategy\edit-strategy\components\edit-risk-per-trade\models

Ø=ÜÄ features\strategy\edit-strategy\components\edit-risk-per-trade\models\risk-per-trade.model.ts

```ts
export const currencies = [
  { code: 'USD', country: 'US' },
  { code: 'EUR', country: 'EU' },
  { code: 'JPY', country: 'JP' },
  { code: 'GBP', country: 'GB' },
  { code: 'AUD', country: 'AU' },
  { code: 'CAD', country: 'CA' },
  { code: 'CHF', country: 'CH' },
  { code: 'CNY', country: 'CN' },
  { code: 'HKD', country: 'HK' },
  { code: 'NZD', country: 'NZ' },
];

```

Ø=ÜÁ features\strategy\edit-strategy\components\edit-risk-reward

Ø=ÜÄ features\strategy\edit-strategy\components\edit-risk-reward\edit-risk-reward.component.ts

```ts
import { CommonModule } from '@angular/common';
import { Component, OnInit } from '@angular/core';
import {
  MaxDailyTradesConfig,
  RiskRewardConfig,
  RuleType,
} from '../../../models/strategy.model';
import { Store } from '@ngrx/store';
import { SettingsService } from '../../../service/strategy.service';
import { riskReward } from '../../../store/strategy.selectors';
import { setRiskRewardConfig } from '../../../store/strategy.actions';

@Component({
  selector: 'app-edit-risk-reward-ratio',
  templateUrl: './edit-risk-reward.component.html',
  styleUrls: ['./edit-risk-reward.component.scss'],
  imports: [CommonModule],
  standalone: true,
})
```

```
export class EditRiskRewardComponent implements OnInit {
  config: RiskRewardConfig = {
    isActive: false,
    riskRewardRatio: '1:2',
    type: RuleType.RISK_REWARD_RATIO,
  };

  inputFirstRatioValue: number = 0;

  inputSecondRatioValue: number = 0;

  initialRatio: string | undefined;

  constructor(private store: Store, private settingsService: SettingsService) {}

  ngOnInit(): void {
    this.listenRuleConfiguration();
  }
  onToggleActive(event: Event) {
    const isActive = (event.target as HTMLInputElement).checked;
    const newConfig = {
      ...this.config,
      isActive: isActive,
      // Reiniciar a 1:2 cuando se desactiva
      riskRewardRatio: isActive ? this.config.riskRewardRatio : '1:2',
    };
    this.updateConfig(newConfig);
  }

  onChangeValue(event: Event, isFirst: boolean) {
    const numValue = Number((event.target as HTMLInputElement).value);
    const numberArray = this.config.riskRewardRatio
      .split(':')
      .map((number) => parseInt(number, 10));

    if (isFirst) {
      numberArray[0] = numValue;
    } else {
      numberArray[1] = numValue;
    }

    const newConfig: RiskRewardConfig = {
      ...this.config,
      riskRewardRatio: numberArray.join(':'),
    };
    this.updateConfig(newConfig);
  }

  // Métodos para spinner (solo para el segundo número)
  incrementSecondValue() {
    if (this.config.isActive) {
      const numberArray = this.config.riskRewardRatio
        .split(':')
        .map((number) => parseInt(number, 10));

      numberArray[1] = numberArray[1] + 1;

      const newConfig: RiskRewardConfig = {
        ...this.config,
        riskRewardRatio: numberArray.join(':'),
      };
      this.updateConfig(newConfig);
    }
  }

  decrementSecondValue() {
    if (this.config.isActive) {
      const numberArray = this.config.riskRewardRatio
        .split(':')
        .map((number) => parseInt(number, 10));
```

```
      if (numberArray[1] > 2) {
        numberArray[1] = numberArray[1] - 1;

        const newConfig: RiskRewardConfig = {
          ...this.config,
          riskRewardRatio: numberArray.join(':'),
        };
        this.updateConfig(newConfig);
      }
    }
  }

  listenRuleConfiguration() {
    this.store
      .select(riskReward)
      .pipe()
      .subscribe((config) => {
        this.config = config;
        if (!this.initialRatio) {
          this.initialRatio = config.riskRewardRatio;
        }
        const numberArray = config.riskRewardRatio
          .split(':')
          .map((number) => parseInt(number, 10));
        this.inputFirstRatioValue = numberArray[0];
        this.inputSecondRatioValue = numberArray[1];
      });
  }

  private updateConfig(config: RiskRewardConfig) {
    this.store.dispatch(setRiskRewardConfig({ config }));
  }
}

```

Ø=ÜÁ features\strategy\models

Ø=ÜÄ features\strategy\models\strategy.model.ts

```
import { Timestamp } from "firebase/firestore";

/**
 * Enum representing the types of trading rules available in the system.
 *
 * Each rule type corresponds to a specific trading constraint or configuration.
 *
 * @enum {string}
 */
export enum RuleType {
  ASSETS_ALLOWED = 'ASSETS ALLOWED',
  DAYS_ALLOWED = 'DAYS ALLOWED',
  MAX_DAILY_TRADES = 'MAX DAILY TRADES',
  MAX_RISK_PER_TRADE = 'MAX RISK PER TRADE',
  RISK_REWARD_RATIO = 'RISK REWARD RATIO',
  TRADING_HOURS = 'TRADING HOURS',
}

/**
 * Enum representing the days of the week.
 *
 * Used for configuring which days trading is allowed.
 *
```

```
 * @enum {string}
 */
export enum Days {
  MONDAY = 'Monday',
  TUESDAY = 'Tuesday',
  WEDNESDAY = 'Wednesday',
  THURSDAY = 'Thursday',
  FRIDAY = 'Friday',
  SATURDAY = 'Saturday',
  SUNDAY = 'Sunday',
}

/**
 * Interface representing the maximum daily trades configuration.
 *
 * Limits the number of trades that can be executed per day.
 *
 * @interface MaxDailyTradesConfig
 */
export interface MaxDailyTradesConfig {
  isActive: boolean;
  type: RuleType;
  maxDailyTrades: number;
}

/**
 * Interface representing the days allowed for trading configuration.
 *
 * Specifies which days of the week trading is permitted.
 *
 * @interface DaysAllowedConfig
 */
export interface DaysAllowedConfig {
  isActive: boolean;
  type: RuleType;
  tradingDays: string[];
}
/**
 * Interface representing the risk/reward ratio configuration.
 *
 * Defines the minimum risk/reward ratio required for trades (e.g., "1:2").
 *
 * @interface RiskRewardConfig
 */
export interface RiskRewardConfig {
  isActive: boolean;
  riskRewardRatio: string;
  type: RuleType;
}
/**
 * Interface representing the maximum risk per trade configuration.
 *
 * Defines how much risk can be taken per trade, with options for:
 * - Review type: MAX (maximum allowed) or FIXED (fixed amount)
 * - Number type: PERCENTAGE or MONEY
 * - Percentage type: INITIAL_B (initial balance), ACTUAL_B (actual balance), or NULL
 *
 * @interface RiskPerTradeConfig
 */
export interface RiskPerTradeConfig {
  isActive: boolean;
  review_type: "MAX" | "FIXED";
  number_type: "PERCENTAGE" | "MONEY";
  percentage_type: "INITIAL_B" | "ACTUAL_B" | "NULL";
  risk_ammount: number;
  balance: number;
  actualBalance?: any;
  type: RuleType;
}
/**
```

```
 * Interface representing the trading hours configuration.
 *
 * Defines the time window during which trading is allowed, including
 * timezone information.
 *
 * @interface HoursAllowedConfig
 */
export interface HoursAllowedConfig {
  isActive: boolean;
  tradingOpenTime: string;
  tradingCloseTime: string;
  timezone: string;
  type: RuleType;
}

/**
 * Interface representing the assets allowed for trading configuration.
 *
 * Specifies which trading instruments/assets are permitted for trading.
 *
 * @interface AssetsAllowedConfig
 */
export interface AssetsAllowedConfig {
  isActive: boolean;
  assetsAllowed: string[];
  type: RuleType;
}

/**
 * Interface representing the complete strategy state (all trading rules).
 *
 * This interface contains all six trading rule configurations that make up
 * a complete trading strategy. It is stored in the 'configurations' collection
 * in Firebase, containing only the rules themselves.
 *
 * Used in:
 * - StrategyComponent: Managing strategy configurations
 * - SettingsService: CRUD operations for strategies
 * - NgRx Store: Local state management
 *
 * @interface StrategyState
 */
export interface StrategyState {
  assetsAllowed: AssetsAllowedConfig;
  hoursAllowed: HoursAllowedConfig;
  riskReward: RiskRewardConfig;
  maxDailyTrades: MaxDailyTradesConfig;
  riskPerTrade: RiskPerTradeConfig;
  daysAllowed: DaysAllowedConfig;
}

/**
 * Interface representing strategy metadata (overview information).
 *
 * This interface contains metadata about a strategy, stored in the
 * 'configuration-overview' collection in Firebase. It includes information
 * like name, status, creation dates, and references to the actual configuration.
 *
 * The actual rules are stored separately in the 'configurations' collection
 * and referenced via configurationId.
 *
 * Features:
 * - Tracks activation/deactivation dates
 * - Supports soft delete (deleted flag)
 * - Links to configuration via configurationId
 *
 * Used in:
 * - StrategyComponent: Displaying strategy cards
 * - SettingsService: Managing strategy metadata
 * - CalendarComponent: Determining strategy compliance for trades
```

```
 *
 * @interface ConfigurationOverview
 */
export interface ConfigurationOverview {
  userId: string;
  name: string;
  status: boolean;
  created_at: any; // Timestamp de Firebase
  updated_at: any; // Timestamp de Firebase
  days_active: number;
  configurationId: string; // ID del documento en la colección 'configurations'
  dateActive?: string[]; // ISO 8601 strings - Fechas cuando se activó la estrategia
  dateInactive?: string[]; // ISO 8601 strings - Fechas cuando se desactivó la estrategia
  deleted?: boolean; // Indica si la estrategia está marcada como eliminada
  deleted_at?: any; // Timestamp de Firebase cuando se marcó como eliminada
}
```

Ø=ÜÁ features\strategy\service

Ø=ÜÄ features\strategy\service\strategy.service.ts

```
import { Injectable } from '@angular/core';
import { MaxDailyTradesConfig, StrategyState, ConfigurationOverview } from '../models/strategy.model';
import { StrategyOperationsService } from '../../../shared/services/strategy-operations.service';
import { AppContextService } from '../../../shared/context';
import { AuthService } from '../../../shared/services/auth.service';

/**
 * Service for managing trading strategies.
 *
 * This service acts as a facade for strategy operations, providing methods
 * to create, read, update, and delete strategies. It manages both the
 * strategy metadata (ConfigurationOverview) and the actual rules (StrategyState).
 *
 * The service uses a two-collection approach:
 * - 'configuration-overview': Stores metadata (name, status, dates)
 * - 'configurations': Stores actual trading rules
 *
 * Responsibilities:
 * - Creating complete strategies (overview + configuration)
 * - Fetching strategies with their configurations
 * - Updating strategy metadata and rules
 * - Activating/deactivating strategies
 * - Soft deleting strategies
 * - Managing user strategy counts
 *
 * Relations:
 * - StrategyOperationsService: Direct Firebase operations
 * - AppContextService: Global state management
 * - AuthService: User count updates
 *
 * @injectable
 * @providedIn root
 */
@Injectable({ providedIn: 'root' })
export class SettingsService {
  /**
   * Constructor for SettingsService.
   *
   * @param strategyOperationsService - Service for direct Firebase operations
   * @param appContext - Application context service for global state
   * @param authService - Authentication service for user operations
   */
```

```
  constructor(
    private strategyOperationsService: StrategyOperationsService,
    private appContext: AppContextService,
    private authService: AuthService
  ) {}

  // ===== CONFIGURATION-OVERVIEW (colección de metadatos) =====

  // Crear configuration-overview (solo metadatos)
  async createConfigurationOverview(userId: string, name: string): Promise<string> {
    return this.strategyOperationsService.createConfigurationOverview(userId, name);
  }

  // Obtener configuration-overview por ID (solo metadatos)
  async getConfigurationOverview(overviewId: string): Promise<ConfigurationOverview | null> {
    return this.strategyOperationsService.getConfigurationOverview(overviewId);
  }

  // Actualizar configuration-overview
  async updateConfigurationOverview(overviewId: string, updates:
Partial<ConfigurationOverview>): Promise<void> {
    return this.strategyOperationsService.updateConfigurationOverview(overviewId, updates);
  }

  // Eliminar configuration-overview
  async deleteConfigurationOverview(overviewId: string): Promise<void> {
    return this.strategyOperationsService.deleteConfigurationOverview(overviewId);
  }

  // ===== CONFIGURATIONS (colección de reglas) =====

  // Crear configuración (solo reglas + IDs)
  async createConfiguration(userId: string, configurationOverviewId: string, configuration:
StrategyState): Promise<void> {
    return this.strategyOperationsService.createConfiguration(userId,
configurationOverviewId, configuration);
  }

  // Obtener configuración por userId
  async getConfiguration(userId: string): Promise<StrategyState | null> {
    return this.strategyOperationsService.getConfiguration(userId);
  }

  // Actualizar configuración
  async updateConfiguration(userId: string, configuration: StrategyState): Promise<void> {
    return this.strategyOperationsService.updateConfiguration(userId, configuration);
  }

  // ===== MÉTODOS INDIVIDUALES NUEVOS =====

  // Crear solo configuration (sin userId ni configurationOverviewId)
  async createConfigurationOnly(configuration: StrategyState): Promise<string> {
    return this.strategyOperationsService.createConfigurationOnly(configuration);
  }

  // Crear configuration-overview con configurationId
  async createConfigurationOverviewWithConfigId(userId: string, name: string,
configurationId: string, shouldBeActive: boolean = false): Promise<string> {
    return this.strategyOperationsService.createConfigurationOverviewWithConfigId(userId,
name, configurationId, shouldBeActive);
  }

  // Actualizar configuration por ID
  async updateConfigurationById(configurationId: string, configuration: StrategyState):
Promise<void> {
    return this.strategyOperationsService.updateConfigurationById(configurationId,
configuration);
  }

  // ===== MÉTODOS COMBINADOS =====

  // Crear estrategia completa (configurations + configuration-overview)
  async createStrategyView(userId: string, name: string, configuration: StrategyState,
shouldBeActive: boolean = false): Promise<string> {
    this.appContext.setLoading('strategies', true);
    this.appContext.setError('strategies', null);

    try {
```

```
      // 1. Crear configuration primero para obtener el ID
      const configurationId = await this.createConfigurationOnly(configuration);

      // 2. Crear configuration-overview con el configurationId
      const overviewId = await this.createConfigurationOverviewWithConfigId(userId, name,
configurationId, shouldBeActive);
      // 3. Actualizar contexto con la nueva estrategia
      const newStrategy = await this.getConfigurationOverview(overviewId);
      if (newStrategy) {
        this.appContext.addStrategy({ ...newStrategy, id: overviewId });
      }

      // 4. Actualizar conteos del usuario
      await this.authService.updateUserCounts(userId);

      this.appContext.setLoading('strategies', false);
      return overviewId;
    } catch (error) {
      this.appContext.setLoading('strategies', false);
      this.appContext.setError('strategies', 'Error al crear estrategia');
      throw error;
    }
  }

  // Obtener estrategia completa (configuration-overview + configurations)
  async getStrategyView(overviewId: string): Promise<{ overview: ConfigurationOverview;
configuration: StrategyState } | null> {
    // 1. Primero obtener configuration-overview
    const overview = await this.getConfigurationOverview(overviewId);

    if (!overview) {
      return null;
    }

    // 2. Luego obtener configuration usando el configurationId
    const configuration = await this.getConfigurationById(overview.configurationId);

    if (!configuration) {
      return null;
    }

    return { overview, configuration };
  }

  // Obtener configuración por ID
  async getConfigurationById(configurationId: string): Promise<StrategyState | null> {
    return this.strategyOperationsService.getConfigurationById(configurationId);
  }

  // Obtener configuración por configurationOverviewId (método legacy para compatibilidad)
  async getConfigurationByOverviewId(overviewId: string): Promise<StrategyState | null> {
    return this.strategyOperationsService.getConfigurationByOverviewId(overviewId);
  }

  // Actualizar estrategia completa
  async updateStrategyView(overviewId: string, updates: { name?: string; configuration?:
StrategyState; userId?: string }): Promise<void> {
    // 1. Primero actualizar configuration-overview si hay cambios de nombre
    if (updates.name) {
      await this.updateConfigurationOverview(overviewId, { name: updates.name });
    }

    // 2. Luego actualizar configuration si hay cambios de reglas
    if (updates.configuration) {
      // Obtener el configurationId del overview
      const overview = await this.getConfigurationOverview(overviewId);
      if (overview && overview.configurationId) {
        await this.updateConfigurationById(overview.configurationId, updates.configuration);
      }
    }
  }
```

```
183
184   // Obtener todas las estrategias de un usuario
185   async getUserStrategyViews(userId: string): Promise<ConfigurationOverview[]> {
186     this.appContext.setLoading('strategies', true);
187     this.appContext.setError('strategies', null);
188
189     try {
190       const strategies = await this.strategyOperationsService.getUserStrategyViews(userId);
191       this.appContext.setUserStrategies(strategies);
192       this.appContext.setLoading('strategies', false);
193       return strategies;
194     } catch (error) {
195       this.appContext.setLoading('strategies', false);
196       this.appContext.setError('strategies', 'Error al obtener estrategias del usuario');
197       throw error;
198     }
199   }
200
201   // Obtener configuración activa (método legacy para compatibilidad)
202   async getActiveConfiguration(userId: string): Promise<ConfigurationOverview | null> {
203     return this.strategyOperationsService.getActiveConfiguration(userId);
204   }
205
206   // Método legacy para compatibilidad
207   async getStrategyConfig(userId: string) {
208     return await this.getConfiguration(userId);
209   }
210
211   // Método legacy para compatibilidad
212   async saveStrategyConfig(userId: string, configurationOverviewId: string) {
213     // Este método ya no es necesario con la nueva estructura
214     console.warn('saveStrategyConfig is deprecated. Use createConfiguration instead.');
215   }
216
217   // Activar una estrategia
218   async activateStrategyView(userId: string, strategyId: string): Promise<void> {
219     return this.strategyOperationsService.activateStrategyView(userId, strategyId);
220   }
221
222   // Actualizar fechas de activación/desactivación de estrategias
223   async updateStrategyDates(userId: string, strategyId: string, dateActive?: Date,
224 dateInactive?: Date): Promise<void> {
225     return this.strategyOperationsService.updateStrategyDates(userId, strategyId,
226 dateActive, dateInactive);
227   }
228
229   // Eliminar una estrategia
230   async deleteStrategyView(strategyId: string): Promise<void> {
231     return this.strategyOperationsService.deleteStrategyView(strategyId);
232   }
233
234   // Marcar una estrategia como deleted (soft delete)
235   async markStrategyAsDeleted(strategyId: string): Promise<void> {
236     // 1. Obtener el userId de la estrategia antes de marcarla como eliminada
237     const strategy = await this.getConfigurationOverview(strategyId);
238     if (!strategy) {
239       throw new Error('Strategy not found');
240     }
241
242     const userId = strategy.userId;
243
244     // 2. Marcar la estrategia como eliminada
245     await this.strategyOperationsService.markStrategyAsDeleted(strategyId);
246
247     // 3. Actualizar conteos del usuario
248     if (userId) {
249       await this.authService.updateUserCounts(userId);
250     }
251   }
252 }
```

Ø=ÜÁ features\strategy\services

Ø=ÜÄ features\strategy\services\balance-cache.service.ts

```typescript
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';

interface BalanceData {
  balance: number;
  timestamp: number;
  accountId: string;
}

/**
 * Service for caching account balance data.
 *
 * This service provides caching for account balances with a 5-minute expiration.
 * It uses both in-memory cache and localStorage for persistence across page reloads.
 * Includes an Observable for real-time balance updates.
 *
 * Features:
 * - In-memory and localStorage caching
 * - 5-minute cache expiration
 * - Observable for balance changes
 * - Automatic cache cleanup
 *
 * Used in:
 * - StrategyComponent: Caching balances for risk calculations
 * - TradingAccountsComponent: Caching account balances
 *
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class BalanceCacheService {
  private balanceCache = new Map<string, BalanceData>();
  private readonly CACHE_DURATION = 5 * 60 * 1000; // 5 minutos
  private balanceSubject = new BehaviorSubject<number>(0);

  /**
   * Obtener balance desde cache o localStorage
   */
  getBalance(accountId: string): number {
    // 1. Verificar cache en memoria
    const cachedBalance = this.balanceCache.get(accountId);
    if (cachedBalance && this.isCacheValid(cachedBalance.timestamp)) {
      return cachedBalance.balance;
    }

    // 2. Verificar localStorage
    const localStorageKey = `balance_${accountId}`;
    const storedBalance = localStorage.getItem(localStorageKey);
    if (storedBalance) {
      try {
        const balanceData: BalanceData = JSON.parse(storedBalance);
        if (this.isCacheValid(balanceData.timestamp)) {
          // Actualizar cache en memoria
          this.balanceCache.set(accountId, balanceData);
          return balanceData.balance;
        }
      } catch (error) {
        console.warn('Error parsing stored balance:', error);
      }
```

```
    }

    return 0; // Valor por defecto
  }

  /**
   * Guardar balance en cache y localStorage
   */
  setBalance(accountId: string, balance: number): void {
    const balanceData: BalanceData = {
      balance,
      timestamp: Date.now(),
      accountId
    };

    // Guardar en cache en memoria
    this.balanceCache.set(accountId, balanceData);

    // Guardar en localStorage
    const localStorageKey = `balance_${accountId}`;
    localStorage.setItem(localStorageKey, JSON.stringify(balanceData));

    // Emitir cambio
    this.balanceSubject.next(balance);
  }

  /**
   * Verificar si el cache es válido
   */
  private isCacheValid(timestamp: number): boolean {
    return Date.now() - timestamp < this.CACHE_DURATION;
  }

  /**
   * Limpiar cache expirado
   */
  clearExpiredCache(): void {
    const now = Date.now();
    this.balanceCache.forEach((value, key) => {
      if (!this.isCacheValid(value.timestamp)) {
        this.balanceCache.delete(key);
        localStorage.removeItem(`balance_${key}`);
      }
    });
  }

  /**
   * Limpiar todo el cache
   */
  clearAllCache(): void {
    this.balanceCache.clear();
    // Limpiar localStorage
    Object.keys(localStorage).forEach(key => {
      if (key.startsWith('balance_')) {
        localStorage.removeItem(key);
      }
    });
  }

  /**
   * Observable para cambios de balance
   */
  getBalanceObservable(): Observable<number> {
    return this.balanceSubject.asObservable();
  }

  /**
   * Verificar si necesita actualizar el balance
   */
  needsUpdate(accountId: string): boolean {
```

```
    const cachedBalance = this.balanceCache.get(accountId);
    if (!cachedBalance) return true;

    return !this.isCacheValid(cachedBalance.timestamp);
  }
}
```

Ø=ÜÄ features\strategy\services\strategy-cache.service.ts

```
import { Injectable } from '@angular/core';
import { ConfigurationOverview } from '../models/strategy.model';
import { StrategyState } from '../models/strategy.model';

/**
 * Service for caching strategy data in memory.
 *
 * This service provides a centralized cache for storing complete strategy data
 * (both overview and configuration) to avoid redundant Firebase queries.
 * Strategies are cached by their ID for quick access.
 *
 * Features:
 * - In-memory caching of strategies
 * - Cache size tracking
 * - Cache clearing functionality
 *
 * Used in:
 * - StrategyComponent: Caching strategies during initialization
 *
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class StrategyCacheService {
  private strategiesCache: Map<string, { overview: ConfigurationOverview; configuration:
StrategyState }> = new Map();
  /**
   * Almacenar estrategia en el cache
   */
  setStrategy(strategyId: string, overview: ConfigurationOverview, configuration:
StrategyState): void {
    this.strategiesCache.set(strategyId, { overview, configuration });
  }

  /**
   * Obtener estrategia del cache
   */
  getStrategy(strategyId: string): { overview: ConfigurationOverview; configuration:
StrategyState } | null {
    return this.strategiesCache.get(strategyId) || null;
  }

  /**
   * Verificar si el cache está cargado
   */
  isCacheLoaded(): boolean {
    return this.strategiesCache.size > 0;
  }

  /**
   * Limpiar todo el cache
   */
  clearCache(): void {
    this.strategiesCache.clear();
  }

```

```
  /**
   * Obtener todas las estrategias del cache
   */
  getAllStrategies(): Map<string, { overview: ConfigurationOverview; configuration: StrategyState }> {
    return this.strategiesCache;
  }

  /**
   * Obtener el tamaño del cache
   */
  getCacheSize(): number {
    return this.strategiesCache.size;
  }

}
```

Ø=ÜÁ features\strategy\store

Ø=ÜÄ features\strategy\store\strategy.actions.ts

```
import { createAction, props } from '@ngrx/store';
import {
  AssetsAllowedConfig,
  DaysAllowedConfig,
  HoursAllowedConfig,
  MaxDailyTradesConfig,
  RiskPerTradeConfig,
  RiskRewardConfig,
  StrategyState,
} from '../models/strategy.model';

/**
 * Action to set the maximum daily trades configuration.
 *
 * @action setMaxDailyTradesConfig
 */
export const setMaxDailyTradesConfig = createAction(
  '[Strategy] Set Max Daily Trades Config',
  props<{ config: MaxDailyTradesConfig }>()
);

/**
 * Action to set the risk/reward ratio configuration.
 *
 * @action setRiskRewardConfig
 */
export const setRiskRewardConfig = createAction(
  '[Strategy] Set Risk Reward Ratio Config',
  props<{ config: RiskRewardConfig }>()
);

/**
 * Action to set the risk per trade configuration.
 *
 * @action setRiskPerTradeConfig
 */
export const setRiskPerTradeConfig = createAction(
  '[Strategy] Set Risk Per Trade Config',
  props<{ config: RiskPerTradeConfig }>()
);

/**
```

```
 * Action to set the days allowed configuration.
 *
 * @action setDaysAllowedConfig
 */
export const setDaysAllowedConfig = createAction(
  '[Strategy] Set Days Allowed Config',
  props<{ config: DaysAllowedConfig }>()
);

/**
 * Action to set the trading hours configuration.
 *
 * @action setHoursAllowedConfig
 */
export const setHoursAllowedConfig = createAction(
  '[Strategy] Set Hours Allowed Config',
  props<{ config: HoursAllowedConfig }>()
);

/**
 * Action to set the assets allowed configuration.
 *
 * @action setAssetsAllowedConfig
 */
export const setAssetsAllowedConfig = createAction(
  '[Strategy] Set Assets Allowed Config',
  props<{ config: AssetsAllowedConfig }>()
);

/**
 * Action to reset the entire strategy configuration.
 *
 * @action resetConfig
 */
export const resetConfig = createAction(
  '[Strategy] Reset Config',
  props<{ config: StrategyState }>()
);

```

Ø=ÜÄ features\strategy\store\strategy.reducer.ts

```
import { createReducer, on } from '@ngrx/store';
import {
  resetConfig,
  setAssetsAllowedConfig,
  setDaysAllowedConfig,
  setHoursAllowedConfig,
  setMaxDailyTradesConfig,
  setRiskPerTradeConfig,
  setRiskRewardConfig,
} from './strategy.actions';
import { Days, RuleType, StrategyState } from '../models/strategy.model';

/**
 * Initial state for the strategy feature.
 *
 * All rules are inactive by default with empty or zero values.
 *
 * @constant initialStrategyState
 */
export const initialStrategyState: StrategyState = {
  maxDailyTrades: {
    isActive: false,
    maxDailyTrades: 0,
    type: RuleType.MAX_DAILY_TRADES,
```

```
  },
  riskReward: {
    isActive: false,
    riskRewardRatio: '1:2',
    type: RuleType.RISK_REWARD_RATIO,
  },
  riskPerTrade: {
    isActive: false,
    review_type: 'MAX',
    number_type: 'PERCENTAGE',
    percentage_type: 'NULL',
    risk_ammount: 0,
    balance: 0,
    actualBalance: 0,
    type: RuleType.MAX_RISK_PER_TRADE,
  },
  daysAllowed: {
    isActive: false,
    type: RuleType.DAYS_ALLOWED,
    tradingDays: [],
  },
  hoursAllowed: {
    isActive: false,
    tradingOpenTime: '',
    tradingCloseTime: '',
    timezone: '',
    type: RuleType.TRADING_HOURS,
  },
  assetsAllowed: {
    isActive: false,
    type: RuleType.ASSETS_ALLOWED,
    assetsAllowed: [],
  },
};

/**
 * Reducer for managing strategy state.
 *
 * Handles all strategy-related actions and updates the state accordingly.
 * Actions handled:
 * - setMaxDailyTradesConfig: Updates max daily trades rule
 * - setRiskRewardConfig: Updates risk/reward ratio rule
 * - setRiskPerTradeConfig: Updates risk per trade rule
 * - setDaysAllowedConfig: Updates days allowed rule
 * - setHoursAllowedConfig: Updates trading hours rule
 * - setAssetsAllowedConfig: Updates assets allowed rule
 * - resetConfig: Resets entire strategy state
 *
 * @reducer strategyReducer
 */
export const strategyReducer = createReducer(
  initialStrategyState,
  on(setMaxDailyTradesConfig, (state, { config }) => ({
    ...state,
    maxDailyTrades: config,
  })),
  on(setRiskRewardConfig, (state, { config }) => ({
    ...state,
    riskReward: config,
  })),
  on(setRiskPerTradeConfig, (state, { config }) => ({
    ...state,
    riskPerTrade: config,
  })),
  on(setDaysAllowedConfig, (state, { config }) => ({
    ...state,
    daysAllowed: config,
  })),
  on(setHoursAllowedConfig, (state, { config }) => ({
    ...state,
```

```
    hoursAllowed: config,
  })),
  on(setAssetsAllowedConfig, (state, { config }) => ({
    ...state,
    assetsAllowed: config,
  })),
  on(resetConfig, (state, { config }) => ({
    ...config,
  }))
);

```

Ø=ÜÄ features\strategy\store\strategy.selectors.ts

```
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { StrategyState } from '../models/strategy.model';

/**
 * Feature selector for the strategy state.
 *
 * @selector selectStrategy
 */
export const selectStrategy = createFeatureSelector<StrategyState>('strategy');

/**
 * Selector for maximum daily trades configuration.
 *
 * @selector selectMaxDailyTrades
 */
export const selectMaxDailyTrades = createSelector(
  selectStrategy,
  (state) => state.maxDailyTrades
);

/**
 * Selector for risk/reward ratio configuration.
 *
 * @selector riskReward
 */
export const riskReward = createSelector(
  selectStrategy,
  (state) => state.riskReward
);

/**
 * Selector for risk per trade configuration.
 *
 * @selector riskPerTrade
 */
export const riskPerTrade = createSelector(
  selectStrategy,
  (state) => state.riskPerTrade
);

/**
 * Selector for days allowed configuration.
 *
 * @selector daysAllowed
 */
export const daysAllowed = createSelector(
  selectStrategy,
  (state) => state.daysAllowed
);

/**
 * Selector for trading hours configuration.
```

```
 *
 * @selector hoursAllowed
 */
export const hoursAllowed = createSelector(
  selectStrategy,
  (state) => state.hoursAllowed
);

/**
 * Selector for assets allowed configuration.
 *
 * @selector assetsAllowed
 */
export const assetsAllowed = createSelector(
  selectStrategy,
  (state) => state.assetsAllowed
);

/**
 * Selector for all strategy rules.
 *
 * @selector allRules
 */
export const allRules = createSelector(selectStrategy, (state) => state);

```

Ø=ÜÁ features\trading-accounts

Ø=ÜÄ features\trading-accounts\trading-accounts.component.ts

```
import { Store } from '@ngrx/store';
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { LoadingSpinnerComponent } from '../../shared/components/loading-spinner/loading-spinner.component';
import { FormsModule } from '@angular/forms';
import { OverviewService } from '../overview/services/overview.service';
import { AccountsTableComponent } from './components/accounts-table/accounts-table.component';
import { User, UserStatus } from '../overview/models/overview';
import { AuthService } from '../auth/service/authService';
import { AccountComponent } from '../account/account.component';
import { selectUser } from '../auth/store/user.selectios';
import { AccountData } from '../auth/models/userModel';
import { ReportService } from '../report/service/report.service';
import { setUserKey } from '../report/store/report.actions';
import { concatMap, from, catchError, of } from 'rxjs';
import { CreateAccountPopupComponent } from '../../shared/components/create-account-popup/create-account-popup.component';
import { Router } from '@angular/router';
import { PlanLimitationsGuard } from '../../guards/plan-limitations.guard';
import { PlanLimitationModalData } from '../../shared/interfaces/plan-limitation-modal.interface';
import { PlanLimitationModalComponent } from '../../shared/components/plan-limitation-modal/plan-limitation-modal.component';
import { PlanBannerComponent } from '../../shared/components/plan-banner/plan-banner.component';
import { AppContextService } from '../../shared/context';

/**
 * Main component for managing trading accounts.
 *
 * This component provides functionality for viewing, adding, editing, and deleting
 * trading accounts. It also handles plan limitations, account balance fetching,
 * and displays plan banners when users approach or reach account limits.
 *
 * Features:
 * - Display all user trading accounts in a table
 * - Add new trading accounts (with plan limitation checks)
```

```
 * - Edit existing trading accounts
 * - Delete trading accounts with confirmation
 * - Fetch and display account balances from API
 * - Plan limitation detection and warnings
 * - Plan upgrade prompts
 *
 * Relations:
 * - AccountsTableComponent: Displays accounts in table format
 * - CreateAccountPopupComponent: Modal for adding/editing accounts
 * - PlanLimitationModalComponent: Modal for plan upgrade prompts
 * - PlanBannerComponent: Banner for plan limitation warnings
 * - AuthService: Account CRUD operations
 * - ReportService: Fetching account balances and user keys
 * - PlanLimitationsGuard: Checking plan limitations
 * - AppContextService: Global state management
 *
 * @component
 * @selector app-trading-accounts
 * @standalone true
 */
@Component({
  selector: 'app-trading-accounts',
  imports: [
    CommonModule,
    LoadingSpinnerComponent,
    FormsModule,
    AccountsTableComponent,
    CreateAccountPopupComponent,
    PlanLimitationModalComponent,
    PlanBannerComponent,
  ],
  templateUrl: './trading-accounts.component.html',
  styleUrl: './trading-accounts.component.scss',
  standalone: true,
})
export class TradingAccountsComponent {
  user: User | null = null;
  userKey: string = '';
  fromDate: string = '';
  toDate: string = '';

  constructor(
    private store: Store,
    private reportSvc: ReportService,
    private userSvc: AuthService,
    private router: Router,
    private planLimitationsGuard: PlanLimitationsGuard,
    private appContext: AppContextService
  ) {}

  loading = false;
  usersData: AccountData[] = [];

  // Plan detection and modal
  showAddAccountModal = false;
  editMode = false;
  accountToEdit: AccountData | null = null;
  planLimitationModal: PlanLimitationModalData = {
    showModal: false,
    modalType: 'upgrade',
    title: '',
    message: '',
    primaryButtonText: '',
    onPrimaryAction: () => {}
  };

  // Button state
  isAddAccountDisabled = false;

  // Plan detection and banner
```

```
  showPlanBanner = false;
  planBannerMessage = '';
  planBannerType = 'info'; // 'info', 'warning', 'success'

  /**
   * Initializes the component on load.
   *
   * Subscribes to context data and store to get user information,
   * then loads configuration and account data.
   *
   * @memberof TradingAccountsComponent
   */
  ngOnInit(): void {
    // Suscribirse a los datos del contexto
    this.subscribeToContextData();

    this.store.select(selectUser).subscribe((userState) => {
      this.user = userState?.user ?? null;
      this.loadConfig();
    });
  }

  /**
   * Subscribes to data from AppContextService.
   *
   * Listens to changes in:
   * - Current user
   * - User accounts
   * - Loading states
   * - Error states
   *
   * Related to:
   * - AppContextService: Global state management
   *
   * @private
   * @memberof TradingAccountsComponent
   */
  private subscribeToContextData() {
    // Suscribirse a los datos del usuario
    this.appContext.currentUser$.subscribe(user => {
      this.user = user;
    });

    // Suscribirse a las cuentas del usuario
    this.appContext.userAccounts$.subscribe(accounts => {
      this.usersData = accounts;
    });

    // Suscribirse a los estados de carga
    this.appContext.isLoading$.subscribe(loading => {
      this.loading = loading.accounts;
    });

    // Suscribirse a los errores
    this.appContext.errors$.subscribe(errors => {
      if (errors.accounts) {
        console.error('Error en cuentas:', errors.accounts);
      }
    });
  }

  /**
   * Loads initial configuration.
   *
   * Fetches user accounts and checks plan limitations.
   *
   * @memberof TradingAccountsComponent
   */
  loadConfig() {
    this.getUserAccounts();
```

```
    // this.checkPlanLimitations(); // Comentado temporalmente
  }

  /**
   * Fetches all trading accounts for the current user.
   *
   * Uses AuthService to get accounts from Firebase, then fetches
   * balance data for each account. Also checks account limitations
   * after loading.
   *
   * Related to:
   * - AuthService.getUserAccounts(): Fetches accounts from Firebase
   * - getBalanceForAccounts(): Fetches balances for all accounts
   * - checkAccountLimitations(): Checks plan limitations
   *
   * @memberof TradingAccountsComponent
   */
  getUserAccounts() {
    this.userSvc
      .getUserAccounts(this.user?.id || '')
      .then((docSnap) => {

        if (docSnap && docSnap.length > 0) {
          this.usersData = docSnap;
          // Don't set loading = false here, wait for balances to load
          this.getBalanceForAccounts();
        } else {
          this.usersData = [];
          this.loading = false; // No accounts, no balances to load
        }

        // Verificar limitaciones después de cargar las cuentas
        this.checkAccountLimitations();
      })
      .catch((err) => {
        this.loading = false;
        console.error('Error to get the config', err);
      });
  }

  /**
   * Fetches balance data for all accounts.
   *
   * Processes accounts sequentially, fetching user key and balance
   * for each account. Updates account balance property with real-time
   * data from the trading API.
   *
   * Related to:
   * - fetchUserKey(): Gets authentication token for each account
   * - getActualBalance(): Fetches balance from API
   * - ReportService: API communication
   *
   * @memberof TradingAccountsComponent
   */
  getBalanceForAccounts() {
    // Loading is already active from getUserAccounts, don't set it again
    from(this.usersData)
      .pipe(concatMap((account) => this.fetchUserKey(account)))
      .subscribe({
        next: () => {},
        complete: () => {
          this.loading = false; // Only set to false when balances are loaded
        },
        error: (err) => {
          this.loading = false;
          console.error(
            'Error en el proceso de actualización de balances',
            err
          );
        },
```

```
      });
  }

  /**
   * Fetches user authentication key for an account.
   *
   * Authenticates with trading account credentials and stores the key
   * in NgRx store, then fetches the actual balance for the account.
   *
   * Related to:
   * - ReportService.getUserKey(): Authenticates and gets token
   * - Store.dispatch(setUserKey()): Stores token in NgRx
   * - getActualBalance(): Fetches balance after authentication
   *
   * @param account - Trading account data with credentials
   * @returns Observable that completes when balance is fetched
   * @memberof TradingAccountsComponent
   */
  fetchUserKey(account: AccountData) {
    return this.reportSvc
      .getUserKey(
        account.emailTradingAccount,
        account.brokerPassword,
        account.server
      )
      .pipe(
        concatMap((key: string) => {
          this.store.dispatch(setUserKey({ userKey: key }));
          return this.getActualBalance(key, account);
        })
      );
  }

  /**
   * Fetches actual balance for an account from the trading API.
   *
   * Uses the authentication key to fetch balance data and updates
   * the account's balance property.
   *
   * Related to:
   * - ReportService.getBalanceData(): Fetches balance from API
   *
   * @param key - User authentication token
   * @param account - Account to fetch balance for
   * @returns Observable that emits the account with updated balance
   * @memberof TradingAccountsComponent
   */
  getActualBalance(key: string, account: AccountData) {
    return this.reportSvc
      .getBalanceData(account.accountID, key, account.accountNumber)
      .pipe(
        concatMap((balanceData) => {
          // Guardar el balance en la cuenta
          account.balance = balanceData.balance || 0;
          return [account];
        })
      );
  }

  /**
   * Deletes a trading account.
   *
   * Removes the account from Firebase and reloads the account list.
   * Also checks plan limitations after deletion.
   *
   * Related to:
   * - AuthService.deleteAccount(): Deletes account from Firebase
   * - loadConfig(): Reloads accounts after deletion
   * - checkPlanLimitations(): Updates limitation status
   *
```

```
   * @param account - Account to delete
   * @memberof TradingAccountsComponent
   */
  deleteAccount(account: AccountData) {
    this.loading = true;
    this.userSvc
      .deleteAccount(account.id)
      .then(async () => {
        // Reload everything after deletion
        this.loadConfig();
        await this.checkPlanLimitations(); // Check plan limitations after deleting account
        this.usersData = [...this.usersData];
      })
      .catch((err) => {
        this.loading = false;
        console.error('Error deleting account', err);
      });
  }

  /**
   * Handles the add account button click.
   *
   * Checks plan limitations before allowing account creation.
   * If user has reached limit, shows upgrade modal or redirects to plan management.
   * If within limits, opens the add account modal.
   *
   * Related to:
   * - PlanLimitationsGuard.checkAccountCreationWithModal(): Checks if user can create account
   * - Router.navigate(): Redirects to plan management if needed
   *
   * @memberof TradingAccountsComponent
   */
  // Add account functionality
  async onAddAccount() {
    if (!this.user?.id) return;

    const currentAccountCount = this.usersData.length;
    const accessCheck = await this.planLimitationsGuard.checkAccountCreationWithModal(this.user.id, currentAccountCount);
    if (!accessCheck.canCreate) {
      // Check if user has Pro plan with maximum accounts (should disable button)
      const limitations = await this.planLimitationsGuard.checkUserLimitations(this.user.id);
      const isProPlanWithMaxAccounts = limitations.planName.toLowerCase().includes('pro') &&
                                       limitations.maxAccounts === 10 &&
                                       currentAccountCount >= 10;

      if (isProPlanWithMaxAccounts) {
        // Pro plan with maximum accounts: button should be disabled (do nothing)
        return;
      } else {
        // Other plans: redirect to account/plan-management
        this.router.navigate(['/account'], {
          queryParams: { tab: 'plan' }
        });
        return;
      }
    }

    // If within limits, show add account modal
    this.editMode = false;
    this.accountToEdit = null;
    this.showAddAccountModal = true;
  }

  /**
   * Handles editing an existing account.
   *
   * Opens the account creation modal in edit mode with the account data pre-filled.
   *
   * @param account - Account to edit
```

```
   * @memberof TradingAccountsComponent
   */
  // Edit account functionality
  onEditAccount(account: AccountData) {
    this.editMode = true;
    this.accountToEdit = account;
    this.showAddAccountModal = true;
  }

  // Modal methods
  onCloseAddAccountModal() {
    this.showAddAccountModal = false;
    this.editMode = false;
    this.accountToEdit = null;
  }

  onAccountCanceled() {
    this.showAddAccountModal = false;
    this.editMode = false;
    this.accountToEdit = null;
  }

  // Plan limitation modal methods
  onClosePlanLimitationModal() {
    this.planLimitationModal.showModal = false;
  }

  /**
   * Checks account limitations and updates button state.
   *
   * Determines if the "Add Account" button should be disabled based on
   * the user's plan and current account count. Also shows plan banners
   * when approaching or at limits.
   *
   * Related to:
   * - PlanLimitationsGuard.checkUserLimitations(): Gets plan limitations
   * - checkPlanLimitations(): Shows plan banners
   *
   * @memberof TradingAccountsComponent
   */
  // Check account limitations and update button state
  async checkAccountLimitations() {
    if (!this.user?.id) {
      this.isAddAccountDisabled = false; // Always active, will redirect if needed
      return;
    }

    try {
      const currentAccountCount = this.usersData.length;
      const limitations = await this.planLimitationsGuard.checkUserLimitations(this.user.id);

      // Only disable button for Pro plan with maximum accounts (10 accounts)
      const isProPlanWithMaxAccounts = limitations.planName.toLowerCase().includes('pro') &&
                                       limitations.maxAccounts === 10 &&
                                       currentAccountCount >= 10;

      this.isAddAccountDisabled = isProPlanWithMaxAccounts;

      // Show banner based on limitations
      await this.checkPlanLimitations();
    } catch (error) {
      console.error('Error checking account limitations:', error);
      this.isAddAccountDisabled = false; // Default to active
    }
  }

  /**
   * Checks plan limitations and displays appropriate banners.
   *
   * Determines if user needs subscription, is banned/cancelled, or has reached
```

```
   * account limits. Shows warning or info banners accordingly.
   *
   * Related to:
   * - PlanLimitationsGuard.checkUserLimitations(): Gets plan limitations
   *
   * @private
   * @memberof TradingAccountsComponent
   */
  // Plan detection and banner methods
  private async checkPlanLimitations() {
    if (!this.user?.id) {
      this.showPlanBanner = false;
      return;
    }

    try {
      // Get user's plan limitations from the guard
      const limitations = await this.planLimitationsGuard.checkUserLimitations(this.user.id);
      const currentAccountCount = this.usersData.length;

      this.showPlanBanner = false;
      this.planBannerMessage = '';
      this.planBannerType = 'info';

      // If user needs subscription or is banned/cancelled
      if (limitations.needsSubscription || limitations.isBanned || limitations.isCancelled) {
        this.showPlanBanner = true;
        this.planBannerMessage = this.getBlockedMessage(limitations);
        this.planBannerType = 'warning';
        return;
      }

      // Check if user has reached account limit
      if (currentAccountCount >= limitations.maxAccounts) {
        this.showPlanBanner = true;
        this.planBannerMessage = `You've reached the account limit for your
${limitations.planName} plan. Move to a higher plan and keep growing your account.`;
        this.planBannerType = 'warning';
      } else if (currentAccountCount >= limitations.maxAccounts - 1) {
        // Show warning when close to limit
        this.showPlanBanner = true;
        this.planBannerMessage = `You have ${limitations.maxAccounts - currentAccountCount}
accounts left on your current plan. Want more? Upgrade anytime.`;
        this.planBannerType = 'info';
      }
    } catch (error) {
      console.error('Error checking plan limitations:', error);
      this.showPlanBanner = false;
    }
  }

  private getBlockedMessage(limitations: any): 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.';
  }


  onUpgradePlan() {
    this.router.navigate(['/account']);
  }

```

```
  onCloseBanner() {
    this.showPlanBanner = false;
  }

  /**
   * Handles account creation event from popup.
   *
   * Reloads account list and checks plan limitations after a new account is created.
   *
   * @param accountData - Created account data (not used, account already in Firebase)
   * @memberof TradingAccountsComponent
   */
  // Popup event handlers
  async onAccountCreated(accountData: any) {
    // Account is already created in Firebase by the popup component
    // Show loading and reload everything
    this.loading = true;
    this.loadConfig(); // Reload accounts
    await this.checkPlanLimitations();
  }

  async onAccountUpdated(accountData: any) {
    // Account is already updated in Firebase by the popup component
    // Show loading and reload everything
    this.loading = true;
    this.loadConfig(); // Reload accounts
    await this.checkPlanLimitations();
  }

}

```

Ø=ÜÁ features\trading-accounts\components\accounts-table

Ø=ÜÄ features\trading-accounts\components\accounts-table\accounts-table.component.ts

```
import { Component, Input, Output, OnInit, OnDestroy, OnChanges, SimpleChanges,
HostListener, ElementRef, ViewChild } from '@angular/core';
import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
import { User } from '../../../overview/models/overview';
import { EventEmitter } from '@angular/core';
import { Timestamp } from 'firebase/firestore';
import { AccountData } from '../../../auth/models/userModel';
import { ShowConfirmationComponent } from '../show-confirmation/show-confirmation.component';
import { Router } from '@angular/router';
import { Subscription } from 'rxjs';
import { NumberFormatterService } from '../../../../../shared/utils/number-formatter.service';

/**
 * Component for displaying trading accounts in a table format.
 *
 * This component provides a filterable and sortable table for trading accounts.
 * It supports filtering by account name and balance range, sorting by creation date,
 * and includes functionality for editing and deleting accounts.
 *
 * Features:
 * - Search by account name
 * - Filter by balance range (with formatted currency inputs)
 * - Sort by creation date (ascending/descending)
 * - Edit account functionality
 * - Delete account with confirmation
 * - Currency formatting for balances
```

```
 *
 * Relations:
 * - ShowConfirmationComponent: Confirmation modal for account deletion
 * - NumberFormatterService: Currency formatting and parsing
 * - TradingAccountsComponent: Parent component (receives accounts, emits edit/delete events)
 *
 * @component
 * @selector app-accounts-table
 * @standalone true
 */
@Component({
  selector: 'app-accounts-table',
  standalone: true,
  imports: [CommonModule, FormsModule, ShowConfirmationComponent],
  templateUrl: './accounts-table.component.html',
  styleUrls: ['./accounts-table.component.scss'],
})
export class AccountsTableComponent implements OnInit, OnDestroy, OnChanges {
  @Input() accounts: AccountData[] = [];
  @Output() delete = new EventEmitter<AccountData>();
  @Output() edit = new EventEmitter<AccountData>();

  initialMinBalance = 0;
  initialMaxBalance = 1000000;
  showFilter = false;
  sortField: 'createdAt' = 'createdAt';
  sortAsc: boolean = true;
  showConfirmation = false;

  // Applied filter values (separate from input values)
  appliedMinBalance = 0;
  appliedMaxBalance = 1000000;

  // Properties for formatted balance inputs
  minBalanceDisplay: string = '';
  maxBalanceDisplay: string = '';
  minBalanceInput: string = '';
  maxBalanceInput: string = '';

  // Reference to filter container for click outside detection
  @ViewChild('filterContainer') filterContainer?: ElementRef;

  // Balance properties (now handled by parent component)
  private subscriptions: Subscription[] = [];

  constructor(
    private router: Router,
    private numberFormatter: NumberFormatterService
  ) {
    // Initialize inputs as empty to show placeholders
    this.minBalanceInput = '';
    this.maxBalanceInput = '';
    this.minBalanceDisplay = '';
    this.maxBalanceDisplay = '';
  }

  private _searchTerm = '';
  accountToDelete!: AccountData;
  get searchTerm(): string {
    return this._searchTerm;
  }
  set searchTerm(val: string) {
    this._searchTerm = val;
  }

  get filteredUsers(): AccountData[] {
    const lower = this._searchTerm.trim().toLowerCase();

    let result = this.accounts.filter((account) => {
      const matchesSearch = `${account.accountName}`
```

```
        .toLowerCase()
        .includes(lower);

      let matchesMinBalance = (account.balance ?? 0) >= this.appliedMinBalance;
      let matchesMaxBalance = (account.balance ?? 0) <= this.appliedMaxBalance;

      if (account.balance === undefined) {
        matchesMinBalance = true;
        matchesMaxBalance = true;
      }

      return matchesSearch && matchesMinBalance && matchesMaxBalance;
    });

    result = result.sort((a, b) => {
      const fieldA =
        a[this.sortField] instanceof Timestamp
          ? (a[this.sortField] as Timestamp).toDate().getTime()
          : String(a[this.sortField]).toLowerCase();
      const fieldB =
        b[this.sortField] instanceof Timestamp
          ? (b[this.sortField] as Timestamp).toDate().getTime()
          : String(b[this.sortField]).toLowerCase();

      if (fieldA < fieldB) return this.sortAsc ? -1 : 1;
      if (fieldA > fieldB) return this.sortAsc ? 1 : -1;
      return 0;
    });

    return result;
  }


  statusClass(status: string) {
    return status;
  }

  returnClass(returnValue: number) {
    return returnValue >= 0 ? 'green' : 'red';
  }

  openFilter() {
    this.showFilter = !this.showFilter;

    if (this.showFilter) {
      // Load current applied values into inputs
      this.initialMinBalance = this.appliedMinBalance;
      this.initialMaxBalance = this.appliedMaxBalance;

      // Set input values for display
      if (this.initialMinBalance > 0) {
        this.minBalanceInput = this.initialMinBalance.toString();
        this.minBalanceDisplay =
this.numberFormatter.formatCurrencyDisplay(this.initialMinBalance);
      } else {
        this.minBalanceInput = '';
        this.minBalanceDisplay = '';
      }

      if (this.initialMaxBalance > 0 && this.initialMaxBalance !== 1000000) {
        this.maxBalanceInput = this.initialMaxBalance.toString();
        this.maxBalanceDisplay =
this.numberFormatter.formatCurrencyDisplay(this.initialMaxBalance);
      } else {
        this.maxBalanceInput = '';
        this.maxBalanceDisplay = '';
      }
    }
  }

  closeFilter() {
    this.showFilter = false;
```

```
  }

  apply() {
    this.applyFilters();
  }

  applyFilters() {
    // Apply the current input values to the filter
    this.appliedMinBalance = this.initialMinBalance;
    this.appliedMaxBalance = this.initialMaxBalance;
    this.showFilter = false;
  }

  // Host listener to detect clicks outside the filter modal
  @HostListener('document:click', ['$event'])
  onDocumentClick(event: Event): void {
    if (this.showFilter && this.filterContainer) {
      const clickedInside = this.filterContainer.nativeElement.contains(event.target as
Node);if (!clickedInside) {
        this.showFilter = false;
      }
    }
  }

  toggleSort() {
    this.sortAsc = !this.sortAsc;
  }

  getUserDate(date: number): Date {
    return new Date(date);
  }

  deleteAccount(account: AccountData) {
    this.showConfirmation = true;
    this.accountToDelete = account;
  }

  editAccount(account: AccountData) {
    this.edit.emit(account);
  }

  confirmDelete() {
    this.delete.emit(this.accountToDelete);
    this.accountToDelete = {} as AccountData;
    this.showConfirmation = false;
  }

  cancelDelete() {
    this.showConfirmation = false;
    this.accountToDelete = {} as AccountData;
  }

  ngOnInit() {
    // Cargar balances para todas las cuentas
    this.loadAccountBalances();
  }

  ngOnChanges(changes: SimpleChanges) {
    if (changes['accounts']) {
      // Recargar balances cuando cambien las cuentas
      if (this.accounts && this.accounts.length > 0) {
        this.loadAccountBalances();
      }
    }
  }

  ngOnDestroy() {
    // Limpiar todas las suscripciones
    this.subscriptions.forEach(sub => sub.unsubscribe());
  }
```

```
238
239     /**
240      * Carga el balance real para todas las cuentas usando el ReportService
241      * Nota: Los balances se cargan desde el componente padre (trading-accounts)
242      */
243     loadAccountBalances() {
244       // Los balances se cargan desde el componente padre
245       // Este método se mantiene para compatibilidad pero no hace nada
246     }
247
248     // Methods for balance input formatting
249     onMinBalanceInput(event: Event) {
250       const target = event.target as HTMLInputElement;
251       this.minBalanceInput = target.value;
252     }
253
254     onMaxBalanceInput(event: Event) {
255       const target = event.target as HTMLInputElement;
256       this.maxBalanceInput = target.value;
257     }
258
259     onMinBalanceFocus() {
260       // When user focuses, show only the number without formatting for editing
261       if (this.initialMinBalance > 0) {
262         this.minBalanceInput = this.initialMinBalance.toString();
263       } else {
264         this.minBalanceInput = '';
265       }
266     }
267
268     onMaxBalanceFocus() {
269       // When user focuses, show only the number without formatting for editing
270       if (this.initialMaxBalance > 0 && this.initialMaxBalance !== 1000000) {
271         this.maxBalanceInput = this.initialMaxBalance.toString();
272       } else {
273         this.maxBalanceInput = '';
274       }
275     }
276
277     onMinBalanceBlur() {
278       // Convert the value to number
279       const numericValue = this.numberFormatter.parseCurrencyValue(this.minBalanceInput);
280       if (!isNaN(numericValue) && numericValue >= 0) {
281         // Save the unformatted value
282         this.initialMinBalance = numericValue;
283
284         // Show visual format (only for display)
285         this.minBalanceDisplay = this.numberFormatter.formatCurrencyDisplay(numericValue);
286
287         // Update the input to show the visual format
288         this.minBalanceInput = this.minBalanceDisplay;
289       } else {
290         // If not a valid number, clear
291         this.minBalanceInput = '';
292         this.minBalanceDisplay = '';
293         this.initialMinBalance = 0;
294       }
295     }
296
297     onMaxBalanceBlur() {
298       // Convert the value to number
299       const numericValue = this.numberFormatter.parseCurrencyValue(this.maxBalanceInput);
300       if (!isNaN(numericValue) && numericValue >= 0) {
301         // Save the unformatted value
302         this.initialMaxBalance = numericValue;
303
304         // Show visual format (only for display)
305         this.maxBalanceDisplay = this.numberFormatter.formatCurrencyDisplay(numericValue);
306
307         // Update the input to show the visual format
```

```
      this.maxBalanceInput = this.maxBalanceDisplay;
    } else {
      // If not a valid number, clear
      this.maxBalanceInput = '';
      this.maxBalanceDisplay = '';
      this.initialMaxBalance = 0; // Reset to 0 instead of 1000000
    }
  }



}

```

Ø=ÜÁ features\trading-accounts\components\show-confirmation

Ø=ÜÄ features\trading-accounts\components\show-confirmation\show-confirmation.component.ts

```typescript
import { Component, Input, Output } from '@angular/core';
import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
import { User } from '../../../overview/models/overview';
import { EventEmitter } from '@angular/core';
import { Timestamp } from 'firebase/firestore';
import { AccountData } from '../../../auth/models/userModel';

/**
 * Component for displaying account deletion confirmation modal.
 *
 * This component requires the user to type the account name exactly
 * to confirm deletion, providing an extra safety measure against accidental deletions.
 *
 * Features:
 * - Account name input validation
 * - Confirm and cancel actions
 * - Event emitters for parent component handling
 *
 * Relations:
 * - AccountsTableComponent: Uses this component for deletion confirmation
 *
 * @component
 * @selector app-show-confirmation
 * @standalone true
 */
@Component({
  selector: 'app-show-confirmation',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './show-confirmation.component.html',
  styleUrls: ['./show-confirmation.component.scss'],
})
export class ShowConfirmationComponent {
  @Input() account: AccountData | null = null;
  @Output() confirm = new EventEmitter<void>();
  @Output() cancel = new EventEmitter<void>();

  accountNameInput: string = '';

  get accountName(): string {
    return this.account?.accountName ?? '';
  }

```

```
  onConfirm() {
    if (this.accountNameInput.trim() === this.accountName) {
      this.confirm.emit();
    }
  }

  onCancel() {
    this.cancel.emit();
  }
}

```

Ø=ÜÁ features\users-details

Ø=ÜÄ features\users-details\users-details.component.ts

```
import { Store } from '@ngrx/store';
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { LoadingPopupComponent } from '../../shared/pop-ups/loading-pop-up/loading-
popup.component';
import { FormsModule } from '@angular/forms';
import { UserManagementService } from '../../shared/services/user-management.service';
import { UsersTableComponent } from './components/users-table/users-table.component';
import { User, UserStatus } from '../overview/models/overview';
import { Timestamp } from 'firebase/firestore';
import { UserModalComponent } from './components/user-modal/user-modal.component';
import { AuthService } from '../auth/service/authService';
import { AppContextService } from '../../shared/context';
import { AlertService } from '../../shared/services/alert.service';
import { ReasonsService } from '../../shared/services/reasons.service';
import { serverTimestamp } from 'firebase/firestore';

/**
 * Component for managing user details and administrative actions.
 *
 * This component provides administrators with tools to view, manage, and perform
 * administrative actions on users. It displays a table of all users (excluding admins)
 * and allows viewing detailed user information, banning/unbanning users, sending
 * password reset links, and logging users out from all devices.
 *
 * Features:
 * - Display all non-admin users in a table
 * - View detailed user information in a modal
 * - Ban users with reason tracking
 * - Unban users and update ban reason records
 * - Send password reset links
 * - Revoke Firebase tokens (logout everywhere)
 * - Admin-only access control
 *
 * Relations:
 * - UsersTableComponent: Displays the user table and handles user selection
 * - UserModalComponent: Shows detailed user information and action buttons
 * - UserManagementService: Fetches and updates user data
 * - AuthService: Handles password reset and token revocation
 * - ReasonsService: Manages ban reason records
 * - AppContextService: Provides current user data and loading states
 * - AlertService: Shows success/error notifications
 *
 * @component
 * @selector app-users-details
 * @standalone true
 */
@Component({
```

```
  selector: 'app-users-details',
  imports: [
    CommonModule,
    LoadingPopupComponent,
    FormsModule,
    UsersTableComponent,
    UserModalComponent,
  ],
  templateUrl: './users-details.component.html',
  styleUrl: './users-details.component.scss',
  standalone: true,
})
export class UsersDetails {
  topUsers: User[] = [];
  selectedUser: User | null = null;
  constructor(
    private store: Store,
    private userManagementService: UserManagementService,
    private userSvc: AuthService,
    private appContext: AppContextService,
    private alertService: AlertService,
    private reasonsService: ReasonsService
  ) {}

  loading = false;
  usersData: User[] = [];

  ngOnInit(): void {
    // Suscribirse a los datos del contexto
    this.subscribeToContextData();
    this.loadConfig();
  }

  private subscribeToContextData() {
    // Suscribirse a los datos del usuario actual
    this.appContext.currentUser$.subscribe(user => {
      // Verificar si el usuario actual es admin
      if (user && user.isAdmin) {
        // Solo los admins pueden ver esta página
        this.loadUsersData();
      }
    });

    // Suscribirse a los estados de carga
    this.appContext.isLoading$.subscribe(loading => {
      this.loading = loading.user;
    });

    // Suscribirse a los errores
    this.appContext.errors$.subscribe(errors => {
      if (errors.user) {
        console.error('Error en gestión de usuarios:', errors.user);
      }
    });
  }

  loadConfig() {
    this.getUsersData();
  }

  loadUsersData() {
    this.getUsersData();
  }

  async getUsersData() {
    try {
      const allUsers = await this.userManagementService.getAllUsers();
      this.usersData = allUsers.filter((user) => !user.isAdmin);
      this.loading = false;
    } catch (error) {
```

```
      this.loading = false;
      console.error('Error getting users data:', error);
    }
  }

  async onBan(event: { username: string; reason: string }) {
    if (!this.selectedUser) return;
    this.loading = true;
    const userId = String(this.selectedUser.id);
    this.reasonsService
      .createReason(userId, event?.reason || '')
      .then(() => this.userManagementService.updateUser(userId, { status:
UserStatus.BANNED }))
      .then(async ()=> {
        await this.getUsersData();
        const refreshed = await this.userManagementService.getUserById(userId);
        if (refreshed) this.selectedUser = refreshed;
        this.alertService.showSuccess('User banned successfully', 'Ban User');
      })
      .catch((err) => {
        console.error('Error banning user', err);
        this.alertService.showError('Error banning user', 'Ban User');
      })
      .finally(() => (this.loading = false));
  }

  async onUnban(username: string) {
    if (!this.selectedUser) return;
    this.loading = true;
    const userId = String(this.selectedUser.id);
    this.userManagementService
      .updateUser(userId, { status: UserStatus.ACTIVE })
      .then(async () => {
        const openReason = await this.reasonsService.getOpenLatestReason(userId);
        if (openReason?.id) {
          await this.reasonsService.updateReason(userId, openReason.id, { dateUnban:
serverTimestamp() } as any);
        }
      })
      .then(async () => {
        await this.getUsersData();
        const refreshed = await this.userManagementService.getUserById(userId);
        if (refreshed) this.selectedUser = refreshed;
        this.alertService.showSuccess('User unbanned successfully', 'Unban User');
      })
      .catch((err) => {
        console.error('Error unbanning user', err);
        this.alertService.showError('Error unbanning user', 'Unban User');
      })
      .finally(() => (this.loading = false));
  }

  onSendResetLink(email: string) {
    if (!email) {
      this.alertService.showInfo('Email is required', 'Password reset');
      return;
    }
    this.loading = true;
    this.userSvc
      .sendPasswordReset(email)
      .then(() => this.alertService.showSuccess('Reset link sent', 'Password reset'))
      .catch((err) => {
        console.error('Error sending reset link', err);
        this.alertService.showError('Error sending reset link', 'Password reset');
      })
      .finally(() => (this.loading = false));
  }

  onLogoutEverywhere(userId: string) {
    if (!userId) return;
    this.loading = true;
    this.userSvc
```

```
      .deleteLinkToken(userId as any)
      .then(() => this.alertService.showSuccess('Firebase token revoked', 'Logout
everywhere'))
      .catch((err) => {
        console.error('Error revoking token', err);
        this.alertService.showError('Error revoking token', 'Logout everywhere');
      })
      .finally(() => (this.loading = false));
  }
}
```

Ø=ÜÁ features\users-details\components\create-user-role-popup

Ø=ÜÄ features\users-details\components\create-user-role-popup\create-user-role-popup.component.ts

```
import { Component, EventEmitter, Input, Output, OnChanges, SimpleChanges } from '@angular/
core';
import { CommonModule } from '@angular/common';
import { ReactiveFormsModule, FormBuilder, FormGroup, Validators, AbstractControl,
ValidationErrors } from '@angular/forms';
import { TextInputComponent } from '../../../../shared/components/text-input/text-
input.component';
import { PasswordInputComponent } from '../../../../shared/components/password-input/
password-input.component';
import { PhoneInputComponent } from '../../../../shared/components/phone-input/phone-
input.component';
import { BirthdayInputComponent } from '../../../../shared/components/birthday-input/
birthday-input.component';
import { AuthService } from '../../../../shared/services/auth.service';
import { SubscriptionService, Subscription } from '../../../../shared/services/subscription-
service';
import { User, UserStatus } from '../../../overview/models/overview';
import { LinkToken } from '../../../../shared/services/tokens-operations.service';
import { UserCredentials } from '../../../auth/models/userModel';
import { AlertService } from '../../../../shared/services/alert.service';

/**
 * Component for creating new users with role selection (user or admin).
 *
 * This component provides a two-step process for creating new users:
 * 1. Role selection (user or admin)
 * 2. User registration form with validation
 *
 * It handles complete user creation including Firebase Authentication,
 * user document creation, link token generation, and free subscription assignment.
 *
 * Features:
 * - Two-step wizard (role selection !’ form)
 * - Role selection (user or admin)
 * - Form validation (email, password, phone, birthday, name)
 * - Email uniqueness check
 * - Age validation (minimum 18 years)
 * - Phone number validation
 * - Email format validation
 * - Confirmation dialogs before creation
 * - Success state with options to create another or go to list
 * - Automatic free subscription assignment
 *
 * Validation:
 * - Email: Must be unique and valid format
 * - Password: Minimum 8 characters
 * - Phone: Valid international format (10-15 digits)
 * - Birthday: User must be at least 18 years old
 * - Name: Minimum 2 characters for first and last name
 *
 * Relations:
 * - AuthService: Creates user in Firebase Auth and Firestore
 * - SubscriptionService: Creates free subscription for new users
 * - TextInputComponent: Form input component
 * - PasswordInputComponent: Password input with validation
 * - PhoneInputComponent: Phone number input with country code
```

```
 * - BirthdayInputComponent: Date picker for birthday
 * - AlertService: Shows error notifications
 *
 * @component
 * @selector app-create-user-role-popup
 * @standalone true
 */
@Component({
  selector: 'app-create-user-role-popup',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule, TextInputComponent, PasswordInputComponent,
PhoneInputComponent, BirthdayInputComponent],
  templateUrl: './create-user-role-popup.component.html',
  styleUrls: ['./create-user-role-popup.component.scss']
})
export class CreateUserRolePopupComponent implements OnChanges {
  @Input() visible = false;
  @Output() close = new EventEmitter<void>();
  @Output() selectRole = new EventEmitter<'user' | 'admin'>();
  @Output() created = new EventEmitter<void>();

  step: 'role' | 'form' = 'role';
  role: 'user' | 'admin' | null = null;
  form: FormGroup;
  showCancelConfirm = false;
  showSuccess = false;
  showCreateConfirm = false;

  constructor(private fb: FormBuilder, private authService: AuthService, private
subscriptionService: SubscriptionService, private alertService: AlertService) {
    this.form = this.fb.group({
      firstName: ['', [Validators.required, Validators.minLength(2)]],
      lastName: ['', [Validators.required, Validators.minLength(2)]],
      email: ['', [Validators.required, Validators.email, this.emailValidator]],
      password: ['', [Validators.required, Validators.minLength(8)]],
      birthday: ['', [Validators.required, this.ageValidator]],
      phoneNumber: ['', [Validators.required, this.phoneValidator]],
    });
  }

  ngOnChanges(changes: SimpleChanges): void {
    if (changes['visible'] && !this.visible) {
      // Al cerrarse, dejar todo listo para comenzar siempre desde 'role'
      this.step = 'role';
      this.role = null;
      this.form.reset({ firstName: '', lastName: '', email: '', password: '', phoneNumber:
'' });this.showCancelConfirm = false;
      this.showSuccess = false;
    }
  }

  onSelect(role: 'user' | 'admin') {
    this.role = role;
    this.step = 'form';
    this.selectRole.emit(role);
  }

  onCancel() {
    this.showCancelConfirm = true;
  }

  // Cancel confirmation overlay actions
  confirmCancel() {
    this.showCancelConfirm = false;
    this.step = 'role';
    this.role = null;
    this.close.emit();
  }

  keepEditing() {
    this.showCancelConfirm = false;
  }
```

```
  // Pre-confirmation before creating the user
  submitCreateUser() {
    if (!this.form.valid || !this.role) return;
    this.showCreateConfirm = true;
  }



  async confirmCreate() {
    try {
      const email = this.form.value.email;
      const password = this.form.value.password;
      // Crear usuario en Firebase Auth
      const credentials: UserCredentials = { email, password } as { email: string; password: string };
      const existingUser = await this.authService.getUserByEmail(email);
      if (existingUser) {
        this.alertService.showError('This email is already registered. Please use a different email or try logging in.', 'Email Already Registered');
        return;
      }

      const userResponse = await this.authService.register(credentials);
      const userId = userResponse.user.uid;

      // Token
      const token: LinkToken = {
        id: email.split('@')[0] + userId.substring(0, 4),
        userId,
      } as any;

      // User doc
      const user: User = {
        id: userId,
        email,
        tokenId: token.id,
        firstName: this.form.value.firstName,
        lastName: this.form.value.lastName,
        phoneNumber: this.form.value.phoneNumber,
        birthday: this.form.value.birthday,
        best_trade: 0,
        netPnl: 0,
        number_trades: 0,
        profit: 0,
        status: this.role === 'admin' ? UserStatus.ADMIN : UserStatus.ACTIVE,
        strategy_followed: 0,
        subscription_date: new Date().getTime(),
        lastUpdated: new Date().getTime(),
        total_spend: 0,
        isAdmin: this.role === 'admin' ? true : false,
        trading_accounts: 0,
        strategies: 0,
      };

      await this.authService.createUser(user as User);
      await this.authService.createLinkToken(token);

      // Crear suscripción Free
      const freeSubscriptionData: Omit<Subscription, 'id' | 'created_at' | 'updated_at'> = {
        planId: 'Cb1B0tpxdE6AP6eMZDo0',
        status: UserStatus.ACTIVE,
        userId,
      };
      await this.subscriptionService.createSubscription(userId, freeSubscriptionData);

      this.showSuccess = true;
      this.created.emit();
    } catch (e) {
      console.error('Error creating user:', e);
    }
```

```
  }

  keepEditingCreate() {
    this.showCreateConfirm = false;
  }

  // ===== Validators (ported from signup.ts) =====
  private phoneValidator(control: AbstractControl): ValidationErrors | null {
    if (!control.value) return null;
    const phoneRegex = /^[\+]?[1-9][\d]{0,15}$/;
    const cleanPhone = String(control.value).replace(/[\s\-\(\)]/g, '');
    if (!phoneRegex.test(cleanPhone)) {
      return { invalidPhone: true };
    }
    if (cleanPhone.length < 10 || cleanPhone.length > 15) {
      return { invalidPhoneLength: true };
    }
    return null;
  }

  private ageValidator(control: AbstractControl): ValidationErrors | null {
    if (!control.value) return null;
    const today = new Date();
    const birthDate = new Date(control.value);
    let age = today.getFullYear() - birthDate.getFullYear();
    const monthDiff = today.getMonth() - birthDate.getMonth();
    if (monthDiff < 0 || (monthDiff === 0 && today.getDate() < birthDate.getDate())) {
      age--;
    }
    if (age < 18) {
      return { underage: true };
    }
    return null;
  }

  private emailValidator(control: AbstractControl): ValidationErrors | null {
    if (!control.value) return null;
    const emailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
    if (!emailRegex.test(control.value)) {
      return { invalidEmailFormat: true };
    }
    return null;
  }

  successCreateAnother() {
    this.showSuccess = false;
    this.form.reset({ firstName: '', lastName: '', email: '', password: '', phoneNumber:
''});
  }

  successGoToList() {
    this.showSuccess = false;
    this.step = 'role';
    this.role = null;
    this.close.emit();
  }
}
```

Ø=ÜÁ features\users-details\components\user-modal

Ø=ÜÄ features\users-details\components\user-modal\user-modal.component.ts

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { User } from '../../../overview/models/overview';
```

```typescript
import { FormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { Timestamp } from 'firebase/firestore';
import { CountryOption } from '../../../../shared/services/countryService';

/**
 * Component for displaying detailed user information in a modal.
 *
 * This component shows comprehensive user details including personal information,
 * trading statistics, account metrics, and provides action buttons for administrative
 * operations. It includes phone number normalization and country detection from
 * phone numbers.
 *
 * Features:
 * - Display user personal information (name, email, phone, birthday)
 * - Show trading statistics (profit, net PnL, trades, spend)
 * - Calculate and display average order value (AOV)
 * - Phone number country detection
 * - Ban user with reason input
 * - Unban user
 * - Send password reset link
 * - Logout user everywhere (revoke tokens)
 *
 * Phone Number Handling:
 * - Normalizes phone numbers with international dial codes
 * - Detects country from phone number prefix
 * - Supports fallback country list for detection
 *
 * Relations:
 * - UsersDetails: Parent component that handles ban/unban actions
 * - CountryService: Provides country options (imported but uses fallback list)
 *
 * @component
 * @selector app-user-modal
 * @standalone true
 */
@Component({
  selector: 'app-user-modal',
  templateUrl: './user-modal.component.html',
  styleUrls: ['./user-modal.component.scss'],
  imports: [FormsModule, CommonModule],
  standalone: true,
})
export class UserModalComponent {
  @Input() user!: User;
  @Output() close = new EventEmitter<void>();
  @Output() ban = new EventEmitter<{ username: string; reason: string }>();
  @Output() unban = new EventEmitter<string>();
  @Output() sendResetLink = new EventEmitter<string>();
  @Output() logoutEverywhere = new EventEmitter<string>();

  today = new Date();

  usernameToBan = '';
  banReason = '';

  private readonly DEFAULT_DIAL_CODE = '+57';
  private readonly fallbackCountries: { dialCode: string; name: string }[] = [
    // North America
    { dialCode: '+1', name: 'United States/Canada' },
    { dialCode: '+1242', name: 'Bahamas' },
    { dialCode: '+1246', name: 'Barbados' },
    { dialCode: '+1264', name: 'Anguilla' },
    { dialCode: '+1268', name: 'Antigua and Barbuda' },
    { dialCode: '+1284', name: 'British Virgin Islands' },
    { dialCode: '+1340', name: 'US Virgin Islands' },
    { dialCode: '+1345', name: 'Cayman Islands' },
    { dialCode: '+1441', name: 'Bermuda' },
    { dialCode: '+1473', name: 'Grenada' },
    { dialCode: '+1649', name: 'Turks and Caicos Islands' },
```

```
    { dialCode: '+1664', name: 'Montserrat' },
    { dialCode: '+1670', name: 'Northern Mariana Islands' },
    { dialCode: '+1671', name: 'Guam' },
    { dialCode: '+1684', name: 'American Samoa' },
    { dialCode: '+1758', name: 'Saint Lucia' },
    { dialCode: '+1767', name: 'Dominica' },
    { dialCode: '+1784', name: 'Saint Vincent and the Grenadines' },
    { dialCode: '+1809', name: 'Dominican Republic' },
    { dialCode: '+1829', name: 'Dominican Republic' },
    { dialCode: '+1849', name: 'Dominican Republic' },
    { dialCode: '+1868', name: 'Trinidad and Tobago' },
    { dialCode: '+1869', name: 'Saint Kitts and Nevis' },
    { dialCode: '+1876', name: 'Jamaica' },
    { dialCode: '+1939', name: 'Puerto Rico' },
    // Central America
    { dialCode: '+502', name: 'Guatemala' },
    { dialCode: '+503', name: 'El Salvador' },
    { dialCode: '+504', name: 'Honduras' },
    { dialCode: '+505', name: 'Nicaragua' },
    { dialCode: '+506', name: 'Costa Rica' },
    { dialCode: '+507', name: 'Panama' },
    { dialCode: '+501', name: 'Belize' },
    // South America
    { dialCode: '+54', name: 'Argentina' },
    { dialCode: '+55', name: 'Brazil' },
    { dialCode: '+56', name: 'Chile' },
    { dialCode: '+57', name: 'Colombia' },
    { dialCode: '+58', name: 'Venezuela' },
    { dialCode: '+51', name: 'Peru' },
    { dialCode: '+593', name: 'Ecuador' },
    { dialCode: '+595', name: 'Paraguay' },
    { dialCode: '+598', name: 'Uruguay' },
    { dialCode: '+591', name: 'Bolivia' },
    { dialCode: '+592', name: 'Guyana' },
    { dialCode: '+597', name: 'Suriname' },
    // Europe
    { dialCode: '+34', name: 'Spain' },
    { dialCode: '+351', name: 'Portugal' },
    { dialCode: '+33', name: 'France' },
    { dialCode: '+49', name: 'Germany' },
    { dialCode: '+39', name: 'Italy' },
    { dialCode: '+44', name: 'United Kingdom' },
    { dialCode: '+31', name: 'Netherlands' },
    { dialCode: '+32', name: 'Belgium' },
    { dialCode: '+352', name: 'Luxembourg' },
    { dialCode: '+41', name: 'Switzerland' },
    { dialCode: '+43', name: 'Austria' },
    { dialCode: '+45', name: 'Denmark' },
    { dialCode: '+46', name: 'Sweden' },
    { dialCode: '+47', name: 'Norway' },
    { dialCode: '+358', name: 'Finland' },
    { dialCode: '+48', name: 'Poland' },
    { dialCode: '+420', name: 'Czech Republic' },
    { dialCode: '+421', name: 'Slovakia' },
    { dialCode: '+36', name: 'Hungary' },
    { dialCode: '+386', name: 'Slovenia' },
    { dialCode: '+385', name: 'Croatia' },
    { dialCode: '+381', name: 'Serbia' },
    { dialCode: '+382', name: 'Montenegro' },
    { dialCode: '+389', name: 'North Macedonia' },
    { dialCode: '+387', name: 'Bosnia and Herzegovina' },
    { dialCode: '+30', name: 'Greece' },
    { dialCode: '+357', name: 'Cyprus' },
    { dialCode: '+353', name: 'Ireland' },
    { dialCode: '+354', name: 'Iceland' },
    { dialCode: '+371', name: 'Latvia' },
    { dialCode: '+370', name: 'Lithuania' },
    { dialCode: '+372', name: 'Estonia' },
    { dialCode: '+375', name: 'Belarus' },
    { dialCode: '+380', name: 'Ukraine' },
```

```
    { dialCode: '+373', name: 'Moldova' },
    { dialCode: '+7', name: 'Russia/Kazakhstan' },
    { dialCode: '+376', name: 'Andorra' },
    { dialCode: '+377', name: 'Monaco' },
    { dialCode: '+378', name: 'San Marino' },
    { dialCode: '+379', name: 'Vatican City' },
    { dialCode: '+423', name: 'Liechtenstein' },
    { dialCode: '+356', name: 'Malta' },
    { dialCode: '+298', name: 'Faroe Islands' },
    { dialCode: '+47', name: 'Svalbard and Jan Mayen' },
    // Africa
    { dialCode: '+20', name: 'Egypt' },
    { dialCode: '+212', name: 'Morocco' },
    { dialCode: '+213', name: 'Algeria' },
    { dialCode: '+216', name: 'Tunisia' },
    { dialCode: '+218', name: 'Libya' },
    { dialCode: '+221', name: 'Senegal' },
    { dialCode: '+225', name: 'Ivory Coast' },
    { dialCode: '+229', name: 'Benin' },
    { dialCode: '+233', name: 'Ghana' },
    { dialCode: '+234', name: 'Nigeria' },
    { dialCode: '+237', name: 'Cameroon' },
    { dialCode: '+240', name: 'Equatorial Guinea' },
    { dialCode: '+241', name: 'Gabon' },
    { dialCode: '+242', name: 'Republic of the Congo' },
    { dialCode: '+243', name: 'Democratic Republic of the Congo' },
    { dialCode: '+244', name: 'Angola' },
    { dialCode: '+248', name: 'Seychelles' },
    { dialCode: '+249', name: 'Sudan' },
    { dialCode: '+250', name: 'Rwanda' },
    { dialCode: '+251', name: 'Ethiopia' },
    { dialCode: '+252', name: 'Somalia' },
    { dialCode: '+253', name: 'Djibouti' },
    { dialCode: '+254', name: 'Kenya' },
    { dialCode: '+255', name: 'Tanzania' },
    { dialCode: '+256', name: 'Uganda' },
    { dialCode: '+257', name: 'Burundi' },
    { dialCode: '+258', name: 'Mozambique' },
    { dialCode: '+260', name: 'Zambia' },
    { dialCode: '+261', name: 'Madagascar' },
    { dialCode: '+263', name: 'Zimbabwe' },
    { dialCode: '+264', name: 'Namibia' },
    { dialCode: '+265', name: 'Malawi' },
    { dialCode: '+266', name: 'Lesotho' },
    { dialCode: '+267', name: 'Botswana' },
    { dialCode: '+268', name: 'Eswatini' },
    { dialCode: '+269', name: 'Comoros' },
    { dialCode: '+27', name: 'South Africa' },
    { dialCode: '+290', name: 'Saint Helena' },
    { dialCode: '+291', name: 'Eritrea' },
    { dialCode: '+220', name: 'Gambia' },
    { dialCode: '+223', name: 'Mali' },
    { dialCode: '+226', name: 'Burkina Faso' },
    { dialCode: '+ Niger', name: 'Niger' },
    { dialCode: '+228', name: 'Togo' },
    { dialCode: '+232', name: 'Sierra Leone' },
    { dialCode: '+231', name: 'Liberia' },
    { dialCode: '+235', name: 'Chad' },
    { dialCode: '+236', name: 'Central African Republic' },
    { dialCode: '+237', name: 'Cameroon' },
    { dialCode: '+238', name: 'Cape Verde' },
    { dialCode: '+239', name: 'São Tomé and Príncipe' },
    { dialCode: '+248', name: 'Seychelles' },
    { dialCode: '+ Mauritania', name: 'Mauritania' },
    { dialCode: '+222', name: 'Mauritania' },
    { dialCode: '+230', name: 'Mauritius' },
    // Middle East / Asia
    { dialCode: '+90', name: 'Turkey' },
    { dialCode: '+30', name: 'Greece' },
    { dialCode: '+972', name: 'Israel' },
```

```
    { dialCode: '+970', name: 'Palestine' },
    { dialCode: '+961', name: 'Lebanon' },
    { dialCode: '+962', name: 'Jordan' },
    { dialCode: '+963', name: 'Syria' },
    { dialCode: '+964', name: 'Iraq' },
    { dialCode: '+965', name: 'Kuwait' },
    { dialCode: '+966', name: 'Saudi Arabia' },
    { dialCode: '+971', name: 'United Arab Emirates' },
    { dialCode: '+973', name: 'Bahrain' },
    { dialCode: '+974', name: 'Qatar' },
    { dialCode: '+968', name: 'Oman' },
    { dialCode: '+98', name: 'Iran' },
    { dialCode: '+92', name: 'Pakistan' },
    { dialCode: '+93', name: 'Afghanistan' },
    { dialCode: '+94', name: 'Sri Lanka' },
    { dialCode: '+95', name: 'Myanmar' },
    { dialCode: '+855', name: 'Cambodia' },
    { dialCode: '+856', name: 'Laos' },
    { dialCode: '+66', name: 'Thailand' },
    { dialCode: '+84', name: 'Vietnam' },
    { dialCode: '+60', name: 'Malaysia' },
    { dialCode: '+62', name: 'Indonesia' },
    { dialCode: '+63', name: 'Philippines' },
    { dialCode: '+65', name: 'Singapore' },
    { dialCode: '+81', name: 'Japan' },
    { dialCode: '+82', name: 'South Korea' },
    { dialCode: '+86', name: 'China' },
    { dialCode: '+852', name: 'Hong Kong' },
    { dialCode: '+853', name: 'Macau' },
    { dialCode: '+886', name: 'Taiwan' },
    { dialCode: '+91', name: 'India' },
    { dialCode: '+880', name: 'Bangladesh' },
    { dialCode: '+977', name: 'Nepal' },
    { dialCode: '+975', name: 'Bhutan' },
    { dialCode: '+960', name: 'Maldives' },
    { dialCode: '+7', name: 'Kazakhstan' },
    { dialCode: '+998', name: 'Uzbekistan' },
    { dialCode: '+996', name: 'Kyrgyzstan' },
    { dialCode: '+992', name: 'Tajikistan' },
    { dialCode: '+993', name: 'Turkmenistan' },
    { dialCode: '+373', name: 'Moldova' },
    // Oceania
    { dialCode: '+61', name: 'Australia' },
    { dialCode: '+64', name: 'New Zealand' },
    { dialCode: '+679', name: 'Fiji' },
    { dialCode: '+682', name: 'Cook Islands' },
    { dialCode: '+683', name: 'Niue' },
    { dialCode: '+685', name: 'Samoa' },
    { dialCode: '+676', name: 'Tonga' },
    { dialCode: '+678', name: 'Vanuatu' },
    { dialCode: '+675', name: 'Papua New Guinea' },
    { dialCode: '+674', name: 'Nauru' },
    { dialCode: '+686', name: 'Kiribati' },
    { dialCode: '+688', name: 'Tuvalu' },
    { dialCode: '+690', name: 'Tokelau' },
    { dialCode: '+689', name: 'French Polynesia' },
    { dialCode: '+687', name: 'New Caledonia' },
    { dialCode: '+681', name: 'Wallis and Futuna' },
];


private normalizePhone(input?: string): string | null {
  if (!input) return null;
  const raw = input
    .toString()
    .trim()
    .replace(/[\s\-().]/g, '');
  if (!raw) return null;
  if (raw.startsWith('+')) return raw;
  if (raw.startsWith('00')) return `+${raw.slice(2)}`;
```

```
    // Sin prefijo internacional, asumimos código por defecto para detección
    return `${this.DEFAULT_DIAL_CODE}${raw}`;
  }

  private resolveCountryFromPhone(phone?: string): string {
    const normalized = this.normalizePhone(phone);
    if (!normalized) return 'Unknown';
    const match = [...this.fallbackCountries]
      .sort((a, b) => b.dialCode.length - a.dialCode.length)
      .find((c) => normalized.startsWith(c.dialCode));
    return match ? match.name : 'Unknown';
  }

  get countryFromPhone(): string {
    return this.resolveCountryFromPhone(this.user?.phoneNumber as unknown as string);
  }

  get aov(): string {
    return this.user.total_spend && this.user.number_trades
      ? (this.user.total_spend / this.user.number_trades).toFixed(2)
      : '0.00';
  }

  onlyNameInitials(user: User) {
    return user.firstName.charAt(0) + user.lastName.charAt(0);
  }
  getUserDate(date: number): Date {
    return new Date(date);
  }
}

```

Ø=ÜÁ features\users-details\components\users-table

Ø=ÜÄ features\users-details\components\users-table\users-table.component.ts

```
import { Component, Input, Output } from '@angular/core';
import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
import { User } from '../../../overview/models/overview';
import { EventEmitter } from '@angular/core';
import { Timestamp } from 'firebase/firestore';
import { CreateUserRolePopupComponent } from '../create-user-role-popup/create-user-role-
popup.component';
/**
 * Component for displaying a filterable and sortable table of users.
 *
 * This component displays users in a paginated table with search, filtering,
 * and sorting capabilities. It also includes functionality to create new users
 * (both regular users and admins) through a popup component.
 *
 * Features:
 * - Search users by name (first name + last name)
 * - Filter by strategy followed percentage range
 * - Sort by first name or last name (ascending/descending)
 * - Pagination with configurable items per page
 * - Status classification (banned, created, active)
 * - Create new user popup (user or admin role)
 * - User selection for detailed view
 *
 * Status Classification:
 * - "banned": User status is explicitly banned
```

```
 * - "created": All user metrics are zero (newly created account)
 * - "active": User has activity (non-zero metrics)
 *
 * Relations:
 * - CreateUserRolePopupComponent: Modal for creating new users
 * - UsersDetails: Parent component that receives selected users
 *
 * @component
 * @selector app-users-table
 * @standalone true
 */
@Component({
  selector: 'app-users-table',
  standalone: true,
  imports: [CommonModule, FormsModule, CreateUserRolePopupComponent],
  templateUrl: './users-table.component.html',
  styleUrls: ['./users-table.component.scss'],
})
export class UsersTableComponent {
  @Input() users: User[] = [];
  @Output() userSelected = new EventEmitter<User>();
  @Output() userCreated = new EventEmitter<void>();

  initialMinStrat = 0;
  initialMaxStrat = 100;
  showFilter = false;
  currentPage: number = 1;
  itemsPerPage: number = 10;
  sortField: 'firstName' | 'lastName' = 'firstName';
  sortAsc: boolean = true;

  showCreateUserPopup = false;

  private _searchTerm = '';
  get searchTerm(): string {
    return this._searchTerm;
  }
  set searchTerm(val: string) {
    this._searchTerm = val;
    this.goToPage(1);
  }

  get filteredUsers(): User[] {
    const lower = this._searchTerm.trim().toLowerCase();

    let result = this.users.filter((user) => {
      const matchesSearch = `${user.firstName} ${user.lastName}`
        .toLowerCase()
        .includes(lower);

      let matchesMinStrat = user.strategy_followed >= this.initialMinStrat;
      let matchesMaxStrat = user.strategy_followed <= this.initialMaxStrat;

      if (user.strategy_followed === undefined) {
        matchesMinStrat = true;
        matchesMaxStrat = true;
      }

      return matchesSearch && matchesMinStrat && matchesMaxStrat;
    });

    result = result.sort((a, b) => {
      const fieldA = a[this.sortField].toLowerCase();
      const fieldB = b[this.sortField].toLowerCase();
      if (fieldA < fieldB) return this.sortAsc ? -1 : 1;
      if (fieldA > fieldB) return this.sortAsc ? 1 : -1;
      return 0;
    });

    return result;
```

```
  }

  get paginatedUsers(): User[] {
    const start = (this.currentPage - 1) * this.itemsPerPage;
    const end = start + this.itemsPerPage;
    return this.filteredUsers.slice(start, end);
  }

  get totalPages(): number {
    return Math.ceil(this.filteredUsers.length / this.itemsPerPage);
  }

  statusClass(user: User): string {
    // Si el status es banned, retornar banned
    if (String(user.status) === 'banned') {
      return 'banned';
    }

    // Verificar si todos los valores están en 0
    const allValuesZero =
      (user.trading_accounts ?? 0) === 0 &&
      (user.strategies ?? 0) === 0 &&
      (user.strategy_followed ?? 0) === 0 &&
      (user.netPnl ?? 0) === 0 &&
      (user.profit ?? 0) === 0 &&
      (user.number_trades ?? 0) === 0 &&
      (user.total_spend ?? 0) === 0;

    // Si todos los valores están en 0, retornar created
    if (allValuesZero) {
      return 'created';
    }

    // Si no todos están en 0, retornar active
    return 'active';
  }

  getDisplayStatus(user: User): string {
    const status = this.statusClass(user);
    return status.charAt(0).toUpperCase() + status.slice(1);
  }

  returnClass(returnValue: number) {
    return returnValue >= 0 ? 'green' : 'red';
  }

  openFilter() {
    this.showFilter = !this.showFilter;
  }

  closeFilter() {
    this.showFilter = false;
  }

  apply() {
    this.showFilter = false;

    this.applyFilters();
  }

  applyFilters() {
    this.goToPage(1);
  }

  resetFilters() {
    this.initialMinStrat = 0;
    this.initialMaxStrat = 100;
    this.applyFilters();
  }

```

```
  goToPage(page: number) {
    if (page < 1) page = 1;
    if (page > this.totalPages) page = this.totalPages;
    this.currentPage = page;
  }

  prevPage() {
    this.goToPage(this.currentPage - 1);
  }

  nextPage() {
    this.goToPage(this.currentPage + 1);
  }

  toggleSort() {
    this.sortAsc = !this.sortAsc;
  }

  getUserDate(date: number): Date {
    return new Date(date);
  }

  emitUser(user: User) {
    this.userSelected.emit(user);
  }

  onOpenCreateUser() {
    this.showCreateUserPopup = true;
  }

  onCloseCreateUser() {
    this.showCreateUserPopup = false;
  }

  onSelectRole(role: 'user' | 'admin') {
    // No cerrar el popup aquí; el componente interno cambia a step 'form'.
  }

  onPopupCreated() {
    this.userCreated.emit();
  }
}
```