TradeSwitch App

Shared TypeScript Code

Generated: 11/25/2025, 6:42:54 PM

Total Files: 53

Table of Contents

10. shared\components\plan-limitation-modal (1 files)
11. shared\components\strategy-card (2 files)
12. shared\components\strategy-guide-modal (1 files)
13. shared\components\subscription-processing (1 files)
14. shared\components\text-input (1 files)
15. shared\context (5 files)
16. shared\interfaces (1 files)
17. shared\mobile-header (1 files)
18. shared\pipes (3 files)
19. shared\pop-ups\alert-popup (1 files)
20. shared\pop-ups\confirm-pop-up (1 files)
21. shared\pop-ups\edit-pop-up (1 files)
22. shared\pop-ups\forgot-password (1 files)
23. shared\pop-ups\loading-pop-up (1 files)
24. shared\pop-ups\stripe-loader-popup (1 files)
25. shared\services (20 files)
26. shared\sidebar-menu (1 files)
27. shared\utils (1 files)

Ø=ÜÁ shared\components

Ø=ÜÄ shared\components\index.ts

```typescript
export { TextInputComponent } from './text-input/text-input.component';
export { PhoneInputComponent } from './phone-input/phone-input.component';
export { BirthdayInputComponent } from './birthday-input/birthday-input.component';
export { PasswordInputComponent } from './password-input/password-input.component';
export { StrategyCardComponent } from './strategy-card/strategy-card.component';
export type { StrategyCardData } from './strategy-card/strategy-card.interface';
export { StrategyGuideModalComponent } from './strategy-guide-modal/strategy-guide-modal.component';
export { LoadingSpinnerComponent } from './loading-spinner/loading-spinner.component';
export { PlanBannerComponent } from './plan-banner/plan-banner.component';

```

Ø=ÜÁ shared\components\birthday-input

Ø=ÜÄ shared\components\birthday-input\birthday-input.component.ts

```typescript
import { Component, Input, forwardRef } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ControlValueAccessor, FormsModule, ReactiveFormsModule, NG_VALUE_ACCESSOR } from '@angular/forms';
interface DateValue {
  month: string;
  day: string;
  year: string;
}

/**
 * Birthday input component with month, day, and year dropdowns.
 *
 * This component provides a date input for birthdays using three separate
 * dropdowns for month, day, and year. It generates valid date ranges and
 * handles date formatting for form submission.
 *
 * Features:
 * - Angular Forms integration (ControlValueAccessor)
 * - Three separate dropdowns (month, day, year)
 * - Year range: 100 years ago to 13 years ago (minimum age validation)
 * - Day range: 1-31 (adjusts based on month/year)
 * - Date formatting (YYYY-MM-DD)
 * - Customizable label
 * - Touch state tracking
 *
 * Date Range:
 * - Years: Current year - 100 to Current year - 13
 * - Days: 1-31 (validated based on month)
 * - Months: January through December
 *
 * Usage:
 * <app-birthday-input
 *   formControlName="birthday"
 *   label="Birthday"
 *   [required]="true">
 * </app-birthday-input>
 *
 * Relations:
 * - Used in registration and profile forms
 * - Integrates with Angular Reactive Forms
 *
```

```
 * @component
 * @selector app-birthday-input
 * @standalone true
 */
@Component({
  selector: 'app-birthday-input',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule],
  templateUrl: './birthday-input.component.html',
  styleUrls: ['./birthday-input.component.scss'],
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => BirthdayInputComponent),
      multi: true
    }
  ]
})
export class BirthdayInputComponent implements ControlValueAccessor {
  @Input() label: string = 'Birthday';
  @Input() required: boolean = false;
  @Input() disabled: boolean = false;

  dateValue: DateValue = { month: '', day: '', year: '' };
  touched: boolean = false;
  showMonthDropdown: boolean = false;
  showDayDropdown: boolean = false;
  showYearDropdown: boolean = false;

  months = [
    { value: '01', label: 'January' },
    { value: '02', label: 'February' },
    { value: '03', label: 'March' },
    { value: '04', label: 'April' },
    { value: '05', label: 'May' },
    { value: '06', label: 'June' },
    { value: '07', label: 'July' },
    { value: '08', label: 'August' },
    { value: '09', label: 'September' },
    { value: '10', label: 'October' },
    { value: '11', label: 'November' },
    { value: '12', label: 'December' }
  ];

  days: string[] = [];
  years: string[] = [];

  onChange = (value: string) => {};
  onTouched = () => {};

  constructor() {
    this.generateDays();
    this.generateYears();
  }

  private generateDays(): void {
    for (let i = 1; i <= 31; i++) {
      this.days.push(i.toString().padStart(2, '0'));
    }
  }

  private generateYears(): void {
    const currentYear = new Date().getFullYear();
    for (let i = currentYear - 100; i <= currentYear - 13; i++) {
      this.years.push(i.toString());
    }
    this.years.reverse();
  }

  writeValue(value: string): void {
```

```
    if (value) {
      const date = new Date(value);
      this.dateValue = {
        month: (date.getMonth() + 1).toString().padStart(2, '0'),
        day: date.getDate().toString().padStart(2, '0'),
        year: date.getFullYear().toString()
      };
    }
  }

  registerOnChange(fn: any): void {
    this.onChange = fn;
  }

  registerOnTouched(fn: any): void {
    this.onTouched = fn;
  }

  setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }

  onDateChange(): void {
    if (this.dateValue.month && this.dateValue.day && this.dateValue.year) {
      const dateString = `${this.dateValue.year}-${this.dateValue.month}-
${this.dateValue.day}`;
      this.onChange(dateString);
    } else {
      this.onChange('');
    }
  }

  onBlur(): void {
    if (!this.touched) {
      this.touched = true;
      this.onTouched();
    }
  }

  onMonthFocus(): void {
    this.showMonthDropdown = true;
    this.showDayDropdown = false;
    this.showYearDropdown = false;
  }

  onDayFocus(): void {
    this.showMonthDropdown = false;
    this.showDayDropdown = true;
    this.showYearDropdown = false;
  }

  onYearFocus(): void {
    this.showMonthDropdown = false;
    this.showDayDropdown = false;
    this.showYearDropdown = true;
  }

  onMonthBlur(): void {
    this.showMonthDropdown = false;
    this.onBlur();
  }

  onDayBlur(): void {
    this.showDayDropdown = false;
    this.onBlur();
  }

  onYearBlur(): void {
    this.showYearDropdown = false;
    this.onBlur();
  }
```

```
}

```

Ø=ÜÁ shared\components\create-account-popup

Ø=ÜÄ shared\components\create-account-popup\create-account-popup.component.ts

```
import { Component, Input, Output, EventEmitter, OnChanges, SimpleChanges } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { TradeLockerApiService } from '../../services/tradelocker-api.service';
import { AuthService } from '../../../features/auth/service/authService';
import { AccountData } from '../../../features/auth/models/userModel';
import { Timestamp } from 'firebase/firestore';
import { NumberFormatterService } from '../../utils/number-formatter.service';

/**
 * Component for creating and editing trading accounts.
 *
 * This component provides a modal interface for adding new trading accounts
 * or editing existing ones. It validates account credentials with TradeLocker
 * API and handles account creation/update operations.
 *
 * Features:
 * - Create new trading accounts
 * - Edit existing trading accounts
 * - Account validation with TradeLocker API
 * - Formatted balance input with currency formatting
 * - Confirmation modals (cancel, success)
 * - Form validation
 * - Account credential validation
 *
 * Account Fields:
 * - Account name
 * - Broker
 * - Email (trading account email)
 * - Password (broker password)
 * - Server
 * - Account ID
 * - Account number
 * - Initial balance (formatted currency input)
 *
 * Validation:
 * - Validates account exists in TradeLocker before creation
 * - Checks account credentials with TradeLocker API
 * - Form field validation
 *
 * Relations:
 * - AuthService: Creates/updates accounts in Firebase
 * - TradeLockerApiService: Validates account credentials
 * - NumberFormatterService: Formats balance input
 *
 * @component
 * @selector app-create-account-popup
 * @standalone true
 */
@Component({
  selector: 'app-create-account-popup',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './create-account-popup.component.html',
  styleUrls: ['./create-account-popup.component.scss']
})
```

```
export class CreateAccountPopupComponent implements OnChanges {
  @Input() visible = false;
  @Input() userId: string = '';
  @Input() currentAccountCount: number = 0;
  @Input() editMode = false;
  @Input() accountToEdit: AccountData | null = null;
  @Output() close = new EventEmitter<void>();
  @Output() create = new EventEmitter<any>();
  @Output() update = new EventEmitter<any>();
  @Output() cancel = new EventEmitter<void>();

  constructor(
    private authService: AuthService,
    private tradeLockerApiService: TradeLockerApiService,
    private numberFormatter: NumberFormatterService
  ) {}

  ngOnChanges(changes: SimpleChanges) {
    // When the popup becomes visible, reset or populate the form
    if (changes['visible'] && changes['visible'].currentValue === true) {
      if (this.editMode && this.accountToEdit) {
        this.populateFormForEdit();
      } else {
        this.resetForm();
      }
    }
  }


  // Form data
  newAccount = {
    emailTradingAccount: '',
    brokerPassword: '',
    accountName: '',
    broker: '',
    server: '',
    accountID: '',
    initialBalance: 0,
    accountNumber: 0,
  };

  // Properties for formatted balance input
  initialBalanceInput: string = '';
  initialBalanceDisplay: string = '';

  // Confirmation modals
  showCancelConfirm = false;
  showSuccessModal = false;

  onClose() {
    this.close.emit();
  }

  onCancel() {
    this.showCancelConfirm = true;
  }

  onConfirmCancel() {
    this.showCancelConfirm = false;
    this.cancel.emit();
  }

  onContinueEditing() {
    this.showCancelConfirm = false;
  }



  async onCreate() {
    // Basic validation
```

```
    if (!this.newAccount.accountName || !this.newAccount.broker ||
        !this.newAccount.emailTradingAccount || !this.newAccount.brokerPassword ||
        !this.newAccount.server || !this.newAccount.accountID || !
this.newAccount.accountNumber) {
      alert('Please fill in all required fields.');
      return;
    }

    try {
      // 1. Validate account exists in TradeLocker
      const accountExists = await this.validateAccountInTradeLocker();
      if (!accountExists) {
        alert('Account does not exist in TradeLocker. Please verify the credentials.');
        return;
      }

      // 2. Validate account email and ID uniqueness
      const validationResult = await this.validateAccountUniqueness();
      if (!validationResult.isValid) {
        alert(validationResult.message);
        return;
      }

      if (this.editMode && this.accountToEdit) {
        // Update existing account
        await this.updateAccount();
      } else {
        // Create new account
        await this.createNewAccount();
      }

    } catch (error) {
      console.error('Error processing trading account:', error);
      alert(`Failed to ${this.editMode ? 'update' : 'create'} trading account. Please try
again.`);
    }
  }

  private async createNewAccount() {
    // Create account object for Firebase
    const accountData = this.createAccountObject();

    // Save to Firebase
    await this.authService.createAccount(accountData);

    // Show success modal
    this.showSuccessModal = true;

    // Emit the created account data
    this.create.emit(accountData);
  }

  private async updateAccount() {
    if (!this.accountToEdit) return;

    // Create updated account object
    const updatedAccountData: AccountData = {
      ...this.accountToEdit,
      accountName: this.newAccount.accountName,
      broker: this.newAccount.broker,
      server: this.newAccount.server,
      emailTradingAccount: this.newAccount.emailTradingAccount,
      brokerPassword: this.newAccount.brokerPassword,
      accountID: this.newAccount.accountID,
      accountNumber: this.newAccount.accountNumber,
      initialBalance: this.newAccount.initialBalance,
      netPnl: this.accountToEdit.netPnl || 0,
      profit: this.accountToEdit.profit || 0,
      bestTrade: this.accountToEdit.bestTrade || 0,
    };

    // Update in Firebase
```

```
197       await this.authService.updateAccount(this.accountToEdit.id, updatedAccountData);
198
199       // Show success modal
200       this.showSuccessModal = true;
201
202       // Emit the updated account data
203       this.update.emit(updatedAccountData);
204     }
205
206     onGoToList() {
207       this.showSuccessModal = false;
208       this.close.emit();
209     }
210
211     // Balance input event handlers
212     onInitialBalanceInput(event: Event) {
213       const target = event.target as HTMLInputElement;
214       this.initialBalanceInput = target.value;
215     }
216
217     onInitialBalanceFocus() {
218       // When user focuses, show only the number without formatting for editing
219       if (this.newAccount.initialBalance > 0) {
220         this.initialBalanceInput = this.newAccount.initialBalance.toString();
221       } else {
222         this.initialBalanceInput = '';
223       }
224     }
225
226     onInitialBalanceBlur() {
227       // Convert the value to number
228       const numericValue = this.numberFormatter.parseCurrencyValue(this.initialBalanceInput);
229       if (!isNaN(numericValue) && numericValue >= 0) {
230         // Save the unformatted value
231         this.newAccount.initialBalance = numericValue;
232
233         // Show visual format (only for display)
234         this.initialBalanceDisplay = this.numberFormatter.formatCurrencyDisplay(numericValue);
235
236         // Update the input to show the visual format
237         this.initialBalanceInput = this.initialBalanceDisplay;
238       } else {
239         // If not a valid number, clear
240         this.initialBalanceInput = '';
241         this.initialBalanceDisplay = '';
242         this.newAccount.initialBalance = 0;
243       }
244     }
245
246     resetForm() {
247       this.newAccount = {
248         accountName: '',
249         broker: '',
250         server: '',
251         emailTradingAccount: '',
252         brokerPassword: '',
253         accountID: '',
254         accountNumber: 1, // Default to 1 as suggested
255         initialBalance: 0,
256       };
257
258       // Reset balance input properties
259       this.initialBalanceInput = '';
260       this.initialBalanceDisplay = '';
261     }
262
263     populateFormForEdit() {
264       if (this.accountToEdit) {
265         this.newAccount = {
266           accountName: this.accountToEdit.accountName || '',
```

```
        broker: this.accountToEdit.broker || '',
        server: this.accountToEdit.server || '',
        emailTradingAccount: this.accountToEdit.emailTradingAccount || '',
        brokerPassword: this.accountToEdit.brokerPassword || '',
        accountID: this.accountToEdit.accountID || '',
        accountNumber: this.accountToEdit.accountNumber || 1,
        initialBalance: this.accountToEdit.initialBalance || 0,
      };

      // Set balance input values for display with currency format
      if (this.newAccount.initialBalance > 0) {
        this.initialBalanceDisplay =
this.numberFormatterformatCurrencyDisplay(this.newAccount.initialBalance);
        this.initialBalanceInput = this.initialBalanceDisplay; // Show formatted value
initially
      } else {
        this.initialBalanceInput = '';
        this.initialBalanceDisplay = '';
      }
    }
  }

  private createAccountObject(): 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: this.userId,
      emailTradingAccount: this.newAccount.emailTradingAccount,
      brokerPassword: this.newAccount.brokerPassword,
      broker: this.newAccount.broker,
      server: this.newAccount.server,
      accountName: this.newAccount.accountName,
      accountID: this.newAccount.accountID,
      accountNumber: this.newAccount.accountNumber,
      initialBalance: this.newAccount.initialBalance,
      createdAt: Timestamp.now(),
      netPnl: 0,
      profit: 0,
      bestTrade: 0,
    };
  }

  /**
   * Validates if the account exists in TradeLocker
   * Simply tries to get JWT token - if successful, credentials are valid
   */
  private async validateAccountInTradeLocker(): Promise<boolean> {
    try {
      return await this.tradeLockerApiService.validateAccount({
        email: this.newAccount.emailTradingAccount,
        password: this.newAccount.brokerPassword,
        server: this.newAccount.server
      });
    } catch (error) {
      console.error('Error validating account in TradeLocker:', error);
      return false;
    }
  }

  /**
   * Validates that broker + server + accountId combination is unique across all accounts
   * Returns validation result with appropriate message
   */
  private async validateAccountUniqueness(): Promise<{isValid: boolean, message: string}> {
    try {
      // Check if broker + server + accountId combination already exists
      const accountExists = await this.authService.checkAccountExists(
        this.newAccount.broker,
        this.newAccount.server,
```

```
        this.newAccount.accountID,
        this.userId
      );

      if (accountExists) {
        return {
          isValid: false,
          message: 'This account is already registered. Try with another account or delete the existing trade account it is linked to.'
        };
      }

      return {
        isValid: true,
        message: ''
      };
    } catch (error) {
      return {
        isValid: false,
        message: 'This account is already registered. Try with another account or delete the existing trade account it is linked to.'
      };
    }
  }
}

```

Ø=ÜÁ shared\components\global-alert

Ø=ÜÄ shared\components\global-alert\global-alert.component.ts

---

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Subscription } from 'rxjs';
import { AlertService, AlertConfig } from '../../services/alert.service';
import { AlertPopupComponent } from '../../pop-ups/alert-popup/alert-popup.component';

/**
 * Global alert component that displays alerts throughout the application.
 *
 * This component subscribes to the AlertService and displays alert dialogs
 * whenever an alert is triggered anywhere in the application. It provides
 * a centralized way to show user notifications.
 *
 * Features:
 * - Subscribes to AlertService alert stream
 * - Displays alerts using AlertPopupComponent
 * - Handles alert close and confirm actions
 * - Automatic cleanup on component destroy
 *
 * Usage:
 * Should be included in the root app component template to enable global
 * alert functionality throughout the application.
 *
 * Relations:
 * - AlertService: Source of alert notifications
 * - AlertPopupComponent: Displays the actual alert UI
 *
 * @component
 * @selector app-global-alert
 * @standalone true
 */
@Component({
  selector: 'app-global-alert',
  standalone: true,
```

```
  imports: [CommonModule, AlertPopupComponent],
  template: `
    <app-alert-popup
      [visible]="alertVisible"
      [title]="alertConfig?.title || ''"
      [message]="alertConfig?.message || ''"
      [buttonText]="alertConfig?.buttonText || 'OK'"
      [type]="alertConfig?.type || 'info'"
      (close)="onClose()"
      (confirm)="onConfirm()">
    </app-alert-popup>
  `
})
export class GlobalAlertComponent implements OnInit, OnDestroy {
  alertVisible = false;
  alertConfig: AlertConfig | null = null;
  private subscription: Subscription = new Subscription();

  constructor(private alertService: AlertService) {}

  ngOnInit(): void {
    this.subscription = this.alertService.alert$.subscribe(alert => {
      this.alertVisible = alert.visible;
      this.alertConfig = alert.config;
    });
  }

  ngOnDestroy(): void {
    this.subscription.unsubscribe();
  }

  onClose(): void {
    this.alertService.hideAlert();
  }

  onConfirm(): void {
    this.alertService.hideAlert();
  }
}

```

Ø=ÜÁ shared\components\loading-spinner

Ø=ÜÄ shared\components\loading-spinner\loading-spinner.component.ts

---

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

/**
 * Component for displaying a loading spinner with customizable title and description.
 *
 * This component provides a reusable loading indicator that can be used throughout
 * the application to show that data is being loaded or an operation is in progress.
 *
 * Features:
 * - Customizable title and description text
 * - Reusable across the application
 * - Standalone component (no module dependencies)
 *
 * Relations:
 * - Used by various components to show loading states
 *
 * @component
 * @selector app-loading-spinner
```

```
 * @standalone true
 */
@Component({
  selector: 'app-loading-spinner',
  standalone: true,
  imports: [],
  templateUrl: './loading-spinner.component.html',
  styleUrl: './loading-spinner.component.scss'
})
export class LoadingSpinnerComponent {
  @Input() title: string = 'Loading...';
  @Input() description: string = 'Please wait while we load your data.';
}

```

Ø=ÜÁ shared\components\order-summary

Ø=ÜÄ shared\components\order-summary\order-summary.component.ts

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

/**
 * Interface for order details display.
 *
 * @interface OrderDetails
 */
export interface OrderDetails {
  planName: string;
  price: number;
  currency: string;
  billing: string;
  features: {
    tradingAccounts: number;
    strategies: number;
    consistencyRules: boolean;
    tradingJournal: boolean;
    liveStatistics: boolean;
  };
  discount?: {
    code: string;
    amount: number;
  };
  taxes: number;
  total: number;
}

/**
 * Interface for order summary configuration.
 *
 * @interface OrderSummaryConfig
 */
export interface OrderSummaryConfig {
  context: 'signup' | 'plan-change';
  planName: string;
  price: number;
  userData?: any;
}

/**
 * Component for displaying order summary before payment.
 *
 * This component shows a detailed summary of the selected plan including
```

```
 * price, features, discounts, taxes, and total. It's used in both signup
 * and plan change flows to confirm the order before payment.
 *
 * Features:
 * - Display plan details (name, price, features)
 * - Calculate discounts (10% for new users)
 * - Calculate taxes (21% VAT)
 * - Display total amount
 * - Progress steps indicator
 * - Context-aware messaging (signup vs plan-change)
 *
 * Calculations:
 * - Discount: 10% of subtotal (applied automatically)
 * - Taxes: 21% of (subtotal - discount)
 * - Total: subtotal - discount + taxes
 *
 * Plan Features:
 * - Trading accounts: Varies by plan (Free: 1, Starter: 2, Pro: 6)
 * - Strategies: Varies by plan (Free: 1, Starter: 3, Pro: 8)
 * - All plans include: Consistency rules, Trading journal, Live statistics
 *
 * Relations:
 * - Used in signup flow before payment
 * - Used in plan change flow before payment
 *
 * @component
 * @selector app-order-summary
 * @standalone true
 */
@Component({
  selector: 'app-order-summary',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './order-summary.component.html',
  styleUrl: './order-summary.component.scss'
})
export class OrderSummaryComponent {
  @Input() config: OrderSummaryConfig = {
    context: 'signup',
    planName: 'Free',
    price: 0
  };
  @Output() continue = new EventEmitter<void>();

  get orderDetails(): OrderDetails {
    const subtotal = this.config.price;
    const discountAmount = this.getDiscountAmount(subtotal);
    const taxes = this.calculateTaxes(subtotal - discountAmount);
    const total = subtotal - discountAmount + taxes;

    return {
      planName: this.config.planName,
      price: subtotal,
      currency: 'USD',
      billing: 'Monthly',
      features: {
        tradingAccounts: this.getTradingAccountsCount(this.config.planName),
        strategies: this.getStrategiesCount(this.config.planName),
        consistencyRules: true,
        tradingJournal: true,
        liveStatistics: true
      },
      discount: discountAmount > 0 ? {
        code: '10RSTORDER',
        amount: discountAmount
      } : undefined,
      taxes: taxes,
      total: total
    };
  }
```

```
115
116    private getTradingAccountsCount(planName: string): number {
117      switch (planName.toLowerCase()) {
118        case 'free': return 1;
119        case 'starter': return 2;
120        case 'pro': return 6;
121        default: return 1;
122      }
123    }
124
125    private getStrategiesCount(planName: string): number {
126      switch (planName.toLowerCase()) {
127        case 'free': return 1;
128        case 'starter': return 3;
129        case 'pro': return 8;
130        default: return 1;
131      }
132    }
133
134    private getDiscountAmount(subtotal: number): number {
135      // Aplicar descuento del 10% para nuevos usuarios o cambios de plan
136      return subtotal > 0 ? Math.round(subtotal * 0.1 * 100) / 100 : 0;
137    }
138
139    private calculateTaxes(subtotal: number): number {
140      // Calcular impuestos del 21% (IVA)
141      return Math.round(subtotal * 0.21 * 100) / 100;
142    }
143
144    onContinue(): void {
145      this.continue.emit();
146    }
147
148    getContextTitle(): string {
149      switch (this.config.context) {
150        case 'signup':
151          return 'Review your plan details.';
152        case 'plan-change':
153          return 'Review your new plan details.';
154        default:
155          return 'Review your plan details.';
156      }
157    }
158
159    getProgressSteps(): { step: number; label: string; status: 'completed' | 'active' }[] {
160      switch (this.config.context) {
161        case 'signup':
162          return [
163            { step: 1, label: 'Plan Selection', status: 'completed' },
164            { step: 2, label: 'Payment', status: 'completed' },
165            { step: 3, label: 'Summary', status: 'active' }
166          ];
167        case 'plan-change':
168          return [
169            { step: 1, label: 'Plan Selection', status: 'completed' },
170            { step: 2, label: 'Payment', status: 'completed' },
171            { step: 3, label: 'Summary', status: 'active' }
172          ];
173        default:
174          return [
175            { step: 1, label: 'Plan Selection', status: 'completed' },
176            { step: 2, label: 'Payment', status: 'completed' },
177            { step: 3, label: 'Summary', status: 'active' }
178          ];
179      }
180    }
181  }
182
```

Ø=ÜÁ shared\components\password-input

Ø=ÜÄ shared\components\password-input\password-input.component.ts

---

```typescript
import { Component, Input, forwardRef } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ControlValueAccessor, FormsModule, ReactiveFormsModule, NG_VALUE_ACCESSOR,
Validator, AbstractControl, ValidationErrors, NG_VALIDATORS } from '@angular/forms';
/**
 * Password input component with validation and strength indicator.
 *
 * This component provides a password input field with comprehensive validation
 * and visual strength feedback. It integrates with Angular Forms and includes
 * built-in password strength validation.
 *
 * Features:
 * - Angular Forms integration (ControlValueAccessor and Validator)
 * - Password visibility toggle
 * - Password strength indicator (weak, medium, strong)
 * - Real-time validation feedback
 * - Validation rules:
 *   - Minimum 8 characters
 *   - Must contain uppercase letter
 *   - Must contain lowercase letter
 *   - Must contain number or symbol
 *   - Cannot contain user's name or email
 * - Color-coded strength indicator
 * - Customizable label and placeholder
 *
 * Password Strength:
 * - Weak: Less than 3 validations pass
 * - Medium: 3-4 validations pass
 * - Strong: All 5 validations pass
 *
 * Usage:
 * <app-password-input
 *   formControlName="password"
 *   label="Password"
 *   [showValidation]="true"
 *   [userEmail]="userEmail"
 *   [userName]="userName">
 * </app-password-input>
 *
 * Relations:
 * - Used in registration and password change forms
 * - Integrates with Angular Reactive Forms
 *
 * @component
 * @selector app-password-input
 * @standalone true
 */
@Component({
  selector: 'app-password-input',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule],
  templateUrl: './password-input.component.html',
  styleUrls: ['./password-input.component.scss'],
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => PasswordInputComponent),
      multi: true
    },
    {
      provide: NG_VALIDATORS,
```

```
      useExisting: forwardRef(() => PasswordInputComponent),
      multi: true
    }
  ]
})
export class PasswordInputComponent implements ControlValueAccessor, Validator {
  @Input() label: string = 'Password';
  @Input() placeholder: string = '********************';
  @Input() required: boolean = false;
  @Input() disabled: boolean = false;
  @Input() showValidation: boolean = false;
  @Input() userEmail: string = '';
  @Input() userName: string = '';

  value: string = '';
  touched: boolean = false;
  showPassword: boolean = false;

  // Validaciones de contraseña
  passwordValidations = {
    noNameOrEmail: false,
    minLength: false,
    hasNumberOrSymbol: false,
    hasUppercase: false,
    hasLowercase: false
  };

  get passwordStrength(): string {
    const validCount = Object.values(this.passwordValidations).filter(Boolean).length;
    if (validCount < 3) return 'weak';
    if (validCount < 5) return 'medium';
    return 'strong';
  }

  get strengthColor(): string {
    const strength = this.passwordStrength;
    switch (strength) {
      case 'weak': return '#EF4444';
      case 'medium': return '#F59E0B';
      case 'strong': return '#22C55E';
      default: return '#6B7280';
    }
  }

  onChange = (value: string) => {};
  onTouched = () => {};

  writeValue(value: string): void {
    this.value = value;
  }

  registerOnChange(fn: any): void {
    this.onChange = fn;
  }

  registerOnTouched(fn: any): void {
    this.onTouched = fn;
  }

  setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }

  onInput(event: Event): void {
    const value = (event.target as HTMLInputElement).value;
    this.value = value;
    this.validatePassword(value);
    this.onChange(value);
  }

```

```
  private validatePassword(password: string): void {
    // No puede contener el nombre o email
    this.passwordValidations.noNameOrEmail = !this.containsNameOrEmail(password);

    // Al menos 8 caracteres
    this.passwordValidations.minLength = password.length >= 8;

    // Al menos 1 número o símbolo
    this.passwordValidations.hasNumberOrSymbol = /[0-9!@#$%^&*()_+\-=\[\]{};':"\
\|,.<>\/?]/.test(password);
    // Al menos 1 letra mayúscula
    this.passwordValidations.hasUppercase = /[A-Z]/.test(password);

    // Al menos 1 letra minúscula
    this.passwordValidations.hasLowercase = /[a-z]/.test(password);
  }

  private containsNameOrEmail(password: string): boolean {
    const lowerPassword = password.toLowerCase();
    const lowerName = this.userName.toLowerCase();
    const lowerEmail = this.userEmail.toLowerCase();

    return lowerPassword.includes(lowerName) || lowerPassword.includes(lowerEmail);
  }

  onBlur(): void {
    if (!this.touched) {
      this.touched = true;
      this.onTouched();
    }
  }

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

  // Implementación de Validator
  validate(control: AbstractControl): ValidationErrors | null {
    if (!control.value || !this.showValidation) {
      return null;
    }

    const password = control.value;
    const errors: ValidationErrors = {};

    // Validar que no contenga nombre o email
    if (this.containsNameOrEmail(password)) {
      errors['containsNameOrEmail'] = true;
    }

    // Validar longitud mínima
    if (password.length < 8) {
      errors['minLength'] = true;
    }

    // Validar que tenga número o símbolo
    if (!/[0-9!@#$%^&*()_+\-=\[\]{};':"\\|,.<>\/?]/.test(password)) {
      errors['hasNumberOrSymbol'] = true;
    }

    // Validar que tenga mayúscula
    if (!/[A-Z]/.test(password)) {
      errors['hasUppercase'] = true;
    }

    // Validar que tenga minúscula
    if (!/[a-z]/.test(password)) {
      errors['hasLowercase'] = true;
    }

```

```
    return Object.keys(errors).length > 0 ? errors : null;
  }
}

```

Ø=ÜÁ shared\components\phone-input

Ø=ÜÄ shared\components\phone-input\phone-input.component.ts

```typescript
import { Component, Input, forwardRef, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ControlValueAccessor, FormsModule, ReactiveFormsModule, NG_VALUE_ACCESSOR } from '@angular/forms';
import { CountryOption, CountryService } from '../../services/countryService';

interface CountryCode {
  code: string;
  flag: string;
  dialCode: string;
}

/**
 * Phone number input component with country code selection.
 *
 * This component provides a phone number input field with country code selection
 * dropdown. It loads countries from CountryService and allows users to select
 * a country code, which is then combined with the phone number.
 *
 * Features:
 * - Angular Forms integration (ControlValueAccessor)
 * - Country code dropdown with flags
 * - Searchable country list
 * - Default country selection (Colombia, fallback to US)
 * - Phone number formatting
 * - Combines dial code with phone number
 * - Customizable label and placeholder
 *
 * Usage:
 * <app-phone-input
 *   formControlName="phoneNumber"
 *   label="Phone Number"
 *   [required]="true">
 * </app-phone-input>
 *
 * Relations:
 * - CountryService: Provides country data with dial codes and flags
 * - Used in registration and profile forms
 *
 * @component
 * @selector app-phone-input
 * @standalone true
 */
@Component({
  selector: 'app-phone-input',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule],
  templateUrl: './phone-input.component.html',
  styleUrls: ['./phone-input.component.scss'],
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => PhoneInputComponent),
      multi: true
    }
```

```
  ]
})
export class PhoneInputComponent implements ControlValueAccessor, OnInit {
  @Input() label: string = 'Phone number';
  @Input() placeholder: string = '(000) 00-0000';
  @Input() required: boolean = false;
  @Input() disabled: boolean = false;

  selectedCountry: CountryCode = { code: 'US', flag: 'https://flagcdn.com/us.svg', dialCode: '+1' };
  phoneNumber: string = '';
  showCountryDropdown: boolean = false;
  touched: boolean = false;
  searchTerm: string = '';
  filteredCountries: CountryCode[] = [];
  countries: CountryCode[] = [];
  loading: boolean = true;

  constructor(private countryService: CountryService) {}

  ngOnInit(): void {
    this.loadCountries();
  }

  private loadCountries(): void {
    this.countryService.getCountries().subscribe({
      next: (countries) => {
        this.countries = countries;
        this.filteredCountries = countries;
        this.loading = false;

        // Buscar Colombia por defecto, si no existe usar US
        const colombia = countries.find(c => c.code === 'CO');
        if (colombia) {
          this.selectedCountry = colombia;
        }
      },
      error: (error) => {
        console.error('Error loading countries:', error);
        this.loading = false;
        // Mantener el país por defecto en caso de error
      }
    });
  }

  onChange = (value: string) => {};
  onTouched = () => {};

  writeValue(value: string): void {
    if (value) {
      // Si el valor ya incluye un código de país, extraer solo el número
      const phoneMatch = value.match(/^(\+\d{1,4})\s(.+)$/);
      if (phoneMatch) {
        const [, dialCode, phoneNumber] = phoneMatch;
        // Buscar el país correspondiente al código de marcación
        const country = this.countries.find(c => c.dialCode === dialCode);
        if (country) {
          this.selectedCountry = country;
        }
        this.phoneNumber = phoneNumber;
      } else {
        this.phoneNumber = value;
      }
    } else {
      this.phoneNumber = '';
    }
  }

  registerOnChange(fn: any): void {
    this.onChange = fn;
  }
```

```
  registerOnTouched(fn: any): void {
    this.onTouched = fn;
  }

  setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }

  onPhoneInput(event: Event): void {
    const value = (event.target as HTMLInputElement).value;
    this.phoneNumber = value;
    this.onChange(this.formatPhoneNumber(value));
  }

  private formatPhoneNumber(phoneNumber: string): string {
    if (!phoneNumber) return '';
    return `${this.selectedCountry.dialCode} ${phoneNumber}`;
  }

  onBlur(): void {
    if (!this.touched) {
      this.touched = true;
      this.onTouched();
    }
  }

  selectCountry(country: CountryCode): void {
    this.selectedCountry = country;
    this.showCountryDropdown = false;
    this.onChange(this.formatPhoneNumber(this.phoneNumber));
  }

  toggleCountryDropdown(): void {
    if (!this.disabled && !this.loading) {
      this.showCountryDropdown = !this.showCountryDropdown;
      if (this.showCountryDropdown) {
        this.filteredCountries = this.countries;
        this.searchTerm = '';
      }
    }
  }

  onSearchInput(event: Event): void {
    const value = (event.target as HTMLInputElement).value;
    this.searchTerm = value;
    this.filterCountries();
  }

  private filterCountries(): void {
    if (!this.searchTerm) {
      this.filteredCountries = this.countries;
    } else {
      const searchLower = this.searchTerm.toLowerCase();
      this.filteredCountries = this.countries.filter(country =>
        country.code.toLowerCase().includes(searchLower) ||
        country.dialCode.includes(searchLower)
      );
    }
  }
}
```

Ø=ÜÁ shared\components\plan-banner

Ø=ÜÄ shared\components\plan-banner\plan-banner.component.ts

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

/**
 * Component for displaying plan limitation banners.
 *
 * This component displays informational or warning banners related to plan
 * limitations, such as approaching account/strategy limits or needing to
 * upgrade. It provides actions to upgrade plans or dismiss the banner.
 *
 * Features:
 * - Customizable message and type (info, warning, error)
 * - Upgrade plan action
 * - Dismiss banner action
 * - Visibility control
 *
 * Banner Types:
 * - info: Informational banner (blue)
 * - warning: Warning banner (yellow/orange)
 * - error: Error banner (red)
 *
 * Usage:
 * <app-plan-banner
 *   [visible]="showBanner"
 *   [message]="bannerMessage"
 *   [type]="'warning'"
 *   (upgradePlan)="onUpgradePlan()"
 *   (closeBanner)="onCloseBanner()">
 * </app-plan-banner>
 *
 * Relations:
 * - Used by components that need to display plan limitation warnings
 * - Often used with PlanLimitationsGuard
 *
 * @component
 * @selector app-plan-banner
 * @standalone true
 */
@Component({
  selector: 'app-plan-banner',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './plan-banner.component.html',
  styleUrls: ['./plan-banner.component.scss']
})
export class PlanBannerComponent {
  @Input() visible: boolean = false;
  @Input() message: string = '';
  @Input() type: string = 'info';

  @Output() upgradePlan = new EventEmitter<void>();
  @Output() closeBanner = new EventEmitter<void>();

  onUpgradePlan() {
    this.upgradePlan.emit();
  }

  onCloseBanner() {
    this.closeBanner.emit();
  }
}
```

62

Ø=ÜÁ shared\components\plan-limitation-modal

Ø=ÜÄ shared\components\plan-limitation-modal\plan-limitation-modal.component.ts

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

/**
 * Interface for plan limitation modal data.
 *
 * @interface PlanLimitationModalData
 */
export interface PlanLimitationModalData {
  showModal: boolean;
  modalType: 'upgrade' | 'blocked';
  title: string;
  message: string;
  primaryButtonText: string;
  secondaryButtonText?: string;
  onPrimaryAction: () => void;
  onSecondaryAction?: () => void;
}

/**
 * Component for displaying plan limitation modals.
 *
 * This component displays modals when users hit plan limitations, either
 * requiring an upgrade or being blocked from a feature. It supports two
 * modal types: upgrade (suggests upgrading) and blocked (access denied).
 *
 * Features:
 * - Two modal types: upgrade and blocked
 * - Customizable title, message, and button texts
 * - Primary and secondary actions
 * - Navigation to account page or signup
 * - Close modal functionality
 *
 * Modal Types:
 * - upgrade: User has reached limit, suggests upgrading plan
 * - blocked: User is banned/cancelled, access denied
 *
 * Usage:
 * <app-plan-limitation-modal
 *   [modalData]="limitationModalData"
 *   (closeModal)="onCloseModal()">
 * </app-plan-limitation-modal>
 *
 * Relations:
 * - PlanLimitationsGuard: Generates modal data
 * - Used by components that check plan limitations
 *
 * @component
 * @selector app-plan-limitation-modal
 * @standalone true
 */
@Component({
  selector: 'app-plan-limitation-modal',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './plan-limitation-modal.component.html',
```

```
  styleUrls: ['./plan-limitation-modal.component.scss']
})
export class PlanLimitationModalComponent {
  @Input() modalData: PlanLimitationModalData = {
    showModal: false,
    modalType: 'upgrade',
    title: '',
    message: '',
    primaryButtonText: '',
    onPrimaryAction: () => {}
  };

  @Output() closeModal = new EventEmitter<void>();

  constructor(private router: Router) {}

  onCloseModal(): void {
    this.closeModal.emit();
  }

  onPrimaryAction(): void {
    this.modalData.onPrimaryAction();
    this.onCloseModal();
  }

  onSecondaryAction(): void {
    if (this.modalData.onSecondaryAction) {
      this.modalData.onSecondaryAction();
    }
    this.onCloseModal();
  }

  navigateToAccount(): void {
    this.router.navigate(['/account']);
  }

  navigateToSignup(): void {
    this.router.navigate(['/signup']);
  }
}

```

Ø=ÜÁ shared\components\strategy-card

Ø=ÜÄ shared\components\strategy-card\strategy-card.component.ts

```
import { Component, Input, Output, EventEmitter, HostListener, OnInit, OnDestroy, Inject }
from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { StrategyCardData } from './strategy-card.interface';
import { NumberFormatterService } from '../../utils/number-formatter.service';
import { StrategyDaysUpdaterService } from '../../services/strategy-days-updater.service';
import { interval, Subscription } from 'rxjs';
import { getFirestore, doc, getDoc, updateDoc } from 'firebase/firestore';
import { firebaseApp } from '../../../firebase/firebase.init';
import { isPlatformBrowser } from '@angular/common';
import { PLATFORM_ID } from '@angular/core';

/**
 * Component for displaying a strategy card with details and actions.
 *
 * This component displays a single strategy in card format with its
 * status, statistics, and action buttons. It supports inline name editing,
```

```
 * favorite toggling, and various strategy operations.
 *
 * Features:
 * - Display strategy information (name, status, rules, days active, win rate)
 * - Inline name editing with Firebase sync
 * - Favorite toggle
 * - More options menu (edit, duplicate, delete)
 * - Customize button
 * - Active rules calculation from Firebase
 * - Days active calculation and update
 * - Status indicators (active/inactive)
 * - Win rate visualization
 *
 * Strategy Information:
 * - Name: Editable inline
 * - Status: Active/Inactive indicator
 * - Rules: Count of active rules
 * - Days Active: Calculated from creation date
 * - Win Rate: Percentage display
 * - Favorite: Star indicator
 *
 * Actions:
 * - Edit: Opens strategy editor
 * - Favorite: Toggles favorite status
 * - More Options: Shows menu (edit, duplicate, delete)
 * - Customize: Opens customization
 *
 * Relations:
 * - StrategyDaysUpdaterService: Updates days active
 * - NumberFormatterService: Formats win rate percentage
 * - Used by StrategyComponent for strategy list display
 *
 * @component
 * @selector app-strategy-card
 * @standalone true
 */
@Component({
  selector: 'app-strategy-card',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './strategy-card.component.html',
  styleUrls: ['./strategy-card.component.scss']
})
export class StrategyCardComponent implements OnInit, OnDestroy {
  @Input() strategy: StrategyCardData = {
    id: '',
    name: '',
    status: false,
    lastModified: '',
    rules: 0,
    days_active: 0,
    winRate: 0,
    isFavorite: false,
    created_at: null,
    updated_at: null,
    userId: '',
    configurationId: '',
    dateActive: [],
    dateInactive: []
  };

  @Input() showCustomizeButton: boolean = true;
  @Input() customizeButtonText: string = 'Customize strategy';

  @Output() edit = new EventEmitter<string>();
  @Output() favorite = new EventEmitter<string>();
  @Output() moreOptions = new EventEmitter<string>();
  @Output() customize = new EventEmitter<string>();
  @Output() editStrategy = new EventEmitter<string>();
  @Output() duplicate = new EventEmitter<string>();
```

```
  @Output() delete = new EventEmitter<string>();
  @Output() nameChanged = new EventEmitter<{id: string, newName: string}>();

  showOptionsMenu = false;
  isEditingName = false;
  editingStrategyName = '';
  isSavingName = false;

  private numberFormatter = new NumberFormatterService();
  private updateSubscription?: Subscription;
  private userId?: string;
  private db: any;

  constructor(
    private daysUpdaterService: StrategyDaysUpdaterService,
    @Inject(PLATFORM_ID) private platformId: Object
  ) {
    if (isPlatformBrowser(this.platformId)) {
      this.db = getFirestore(firebaseApp);
    }
  }

  ngOnInit(): void {
    this.calculateActiveRules();
    this.updateDaysActive();
  }

  ngOnDestroy(): void {
    // Limpiar suscripciones si las hay
  }

  /**
   * Actualiza los días activos de la estrategia
   */
  private async updateDaysActive(): Promise<void> {
    if (!this.strategy.id || !this.strategy.userId) {
      return;
    }

    try {
      // Calcular días activos localmente para mostrar inmediatamente
      if (this.strategy.created_at) {
        const calculatedDays =
this.daysUpdaterService.getDaysActive(this.strategy.created_at);
        this.strategy.days_active = calculatedDays;
      }

      // Actualizar en Firebase
      await this.daysUpdaterService.updateStrategyDaysActive(this.strategy.id,
this.strategy.userId);
    } catch (error) {
      console.error('Error updating days active:', error);
    }
  }

  onEdit() {
    this.startEditName();
  }

  startEditName() {
    this.isEditingName = true;
    this.editingStrategyName = this.strategy.name;
    // Focus en el input después de que se renderice
    setTimeout(() => {
      const input = document.querySelector('.strategy-name-input') as HTMLInputElement;
      if (input) {
        input.focus();
        input.select();
      }
    }, 0);
  }

```

```
  async saveStrategyName() {
    if (this.editingStrategyName.trim() && this.editingStrategyName.trim() !== this.strategy.name) {
      const newName = this.editingStrategyName.trim();
      this.isSavingName = true;

      try {
        // Actualizar en Firebase
        await this.updateStrategyNameInFirebase(newName);

        // Actualizar el nombre localmente
        this.strategy.name = newName;

        // Emitir evento para notificar al componente padre
        this.nameChanged.emit({
          id: this.strategy.id,
          newName: newName
        });
      } catch (error) {
        // No actualizamos el nombre local si falla en Firebase
        console.error('Error updating the name of the strategy:', error);
      } finally {
        this.isSavingName = false;
      }
    }

    this.isEditingName = false;
    this.editingStrategyName = '';
  }

  cancelEditName() {
    this.isEditingName = false;
    this.editingStrategyName = '';
  }

  onFavorite() {
    this.favorite.emit(this.strategy.id);
  }

  onMoreOptions(event: Event) {
    event.preventDefault();
    event.stopPropagation();
    this.showOptionsMenu = !this.showOptionsMenu;
    // Removed moreOptions.emit() to prevent Google alert
  }

  onCustomize() {
    this.customize.emit(this.strategy.id);
  }

  onEditStrategy() {
    this.editStrategy.emit(this.strategy.id);
  }

  onDuplicate() {
    this.showOptionsMenu = false;
    this.duplicate.emit(this.strategy.id);
  }

  onDelete() {
    this.showOptionsMenu = false;
    this.delete.emit(this.strategy.id);
  }

  onCloseOptionsMenu() {
    this.showOptionsMenu = false;
  }

  @HostListener('document:click', ['$event'])
  onDocumentClick(event: Event) {
    // Solo cerrar si el click no es en el botón de more options o en el menú
```

```
    const target = event.target as HTMLElement;
    const moreBtn = target.closest('.more-btn');
    const optionsMenu = target.closest('.options-menu');

    if (this.showOptionsMenu && !moreBtn && !optionsMenu) {
      this.showOptionsMenu = false;
    }
  }

  getStatusClass(): string {
    return this.strategy.status ? 'active' : 'inactive';
  }

  getStatusText(): string {
    return this.strategy.status ? 'Active' : 'Inactive';
  }

  getWinRateWidth(): string {
    return `${this.strategy.winRate}%`;
  }

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

  /**
   * Actualiza el nombre de la estrategia en Firebase
   */
  private async updateStrategyNameInFirebase(newName: string): Promise<void> {
    if (!this.db || !this.strategy.id) {
      throw new Error('Cannot update: missing database information or strategy ID');
    }

    try {
      // Intentar actualizar en la colección 'strategies'
      const strategyRef = doc(this.db, 'configuration-overview', this.strategy.id);
      const strategyDoc = await getDoc(strategyRef);

      if (strategyDoc.exists()) {
        // Si existe, actualizar el documento
        await updateDoc(strategyRef, {
          name: newName,
          updated_at: new Date()
        });
      } else {
        // Si no existe, mostrar error específico
        const errorMsg = `The strategy with ID "${this.strategy.id}" does not exist in
Firebase`;
        console.error(errorMsg);
        throw new Error(errorMsg);
      }
    } catch (error) {
      console.error('Error updating the name in Firebase:', error);
      throw error;
    }
  }

  /**
   * Calcula las reglas activas desde la colección configurations
   */
  private async calculateActiveRules(): Promise<void> {
    if (!this.db || !this.strategy.configurationId) {
      return;
    }

    try {
      const configDoc = await getDoc(doc(this.db, 'configurations',
this.strategy.configurationId));
      if (configDoc.exists()) {
        const configData = configDoc.data();
        let activeRulesCount = 0;
```

```
298
299         // Contar reglas activas
300         if (configData['maxDailyTrades']?.isActive) activeRulesCount++;
301         if (configData['riskReward']?.isActive) activeRulesCount++;
302         if (configData['riskPerTrade']?.isActive) activeRulesCount++;
303         if (configData['daysAllowed']?.isActive) activeRulesCount++;
304         if (configData['hoursAllowed']?.isActive) activeRulesCount++;
305         if (configData['assetsAllowed']?.isActive) activeRulesCount++;
306
307         this.strategy.rules = activeRulesCount;
308       }
309     } catch (error) {
310       console.error('Error calculating active rules:', error);
311     }
312   }
313 }
314
```

Ø=ÜÄ shared\components\strategy-card\strategy-card.interface.ts

```
1  /**
2   * Interface for strategy card display data.
3   *
4   * This interface defines the data structure used by StrategyCardComponent
5   * to display strategy information in card format.
6   *
7   * @interface StrategyCardData
8   */
9  export interface StrategyCardData {
10   id: string;
11   name: string;
12   status: boolean; // true/false como en Firebase
13   lastModified: string;
14   rules: number; // Se calculará dinámicamente
15   days_active: number; // Viene de Firebase
16   winRate: number; // Se calculará dinámicamente
17   isFavorite?: boolean;
18   created_at: any; // Timestamp de Firebase
19   updated_at: any; // Timestamp de Firebase
20   userId: string;
21   configurationId: string;
22   dateActive?: string[]; // ISO 8601 strings - Array de fechas cuando se activó la estrategia
23   dateInactive?: string[]; // ISO 8601 strings - Array de fechas cuando se desactivó la
24 estrategia
25 }
```

Ø=ÜÁ shared\components\strategy-guide-modal

Ø=ÜÄ shared\components\strategy-guide-modal\strategy-guide-modal.component.ts

```
1  import { Component, EventEmitter, Output } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3
4  /**
5   * Component for displaying a strategy creation guide modal.
6   *
7   * This component provides a step-by-step guide for new users on how to
8   * create and manage trading strategies. It includes three steps with
9   * images and descriptions, and allows users to skip the guide in the future.
```

```
 *
 * Features:
 * - Three-step guide (Create, Select, Edit)
 * - Step navigation (next/previous)
 * - Image and description for each step
 * - "Don't show again" option
 * - Direct navigation to strategy editing
 * - Close modal functionality
 *
 * Guide Steps:
 * 1. Create a strategy: Design trading plan with custom rules
 * 2. Select strategy: Pick strategy for each account
 * 3. Edit a strategy: Fine-tune rules anytime
 *
 * Usage:
 * Shown to first-time users when they access the strategy page.
 * Users can opt to not see it again.
 *
 * Relations:
 * - Used by StrategyComponent for first-time user guidance
 *
 * @component
 * @selector app-strategy-guide-modal
 * @standalone true
 */
@Component({
  selector: 'app-strategy-guide-modal',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './strategy-guide-modal.component.html',
  styleUrl: './strategy-guide-modal.component.scss'
})
export class StrategyGuideModalComponent {
  @Output() closeModal = new EventEmitter<void>();
  @Output() dontShowAgain = new EventEmitter<void>();
  @Output() editStrategy = new EventEmitter<void>();

  currentStep = 0;
  totalSteps = 3;

  steps = [
    {
      title: 'Create a strategy',
      description: 'Design your trading plan and set personalized rules that fit you.',
      image: 'assets/images/strategy/Create.webp',
      content: 'Create your own trading strategy with custom rules and parameters.'
    },
    {
      title: 'Select strategy',
      description: 'Pick the perfect strategy for each account with ease.',
      image: 'assets/images/strategy/Select.webp',
      content: 'Choose from your created strategies and apply them to your trading accounts.'
    },
    {
      title: 'Edit a strategy',
      description: 'Fine tune rules anytime to activate or disable specific ones.',
      image: 'assets/images/strategy/Edit.webp',
      content: 'Modify your strategies, adjust parameters, and optimize your trading rules.'
    }
  ];

  onNext(): void {
    if (this.currentStep < this.totalSteps - 1) {
      this.currentStep++;
    } else {
      // En el último paso, emitir evento para editar estrategia
      this.editStrategy.emit();
    }
  }

```

```
  onPrevious(): void {
    if (this.currentStep > 0) {
      this.currentStep--;
    }
  }

  onClose(): void {
    this.closeModal.emit();
  }

  onDontShowAgain(): void {
    this.dontShowAgain.emit();
  }

  getCurrentStep() {
    return this.steps[this.currentStep];
  }

  getButtonText(): string {
    return this.currentStep === this.totalSteps - 1 ? 'Edit strategy' : 'Next';
  }
}

```

Ø=ÜÁ shared\components\subscription-processing

Ø=ÜÄ shared\components\subscription-processing\subscription-processing.component.ts

```
import { Component, Input, Output, EventEmitter, OnInit, OnDestroy, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { SubscriptionService } from '../../services/subscription-service';
import { UserStatus } from '../../../features/overview/models/overview';

/**
 * Interface for subscription processing configuration.
 *
 * @interface SubscriptionProcessingConfig
 */
export interface SubscriptionProcessingConfig {
  paymentId: string;
  userId: string;
  context: 'signup' | 'plan-change';
  planName?: string;
}

/**
 * Component for displaying subscription payment processing status.
 *
 * This component shows a processing screen while a subscription payment is
 * being verified. It polls the subscription status and displays success
 * or error states based on the payment result.
 *
 * Features:
 * - Payment status polling (every 2 seconds)
 * - Success state display
 * - Error state display with retry option
 * - Timeout handling (30 seconds)
 * - Context-aware messaging (signup vs plan-change)
 * - Go back functionality
 *
 * Status States:
 * - processing: Payment is being verified
 * - success: Payment confirmed (PURCHASED status)
```

```
 * - error: Payment failed or timeout
 *
 * Polling:
 * - Checks subscription status every 2 seconds
 * - Stops when status changes or timeout occurs
 * - Supports retry on error
 *
 * Relations:
 * - SubscriptionService: Checks subscription status
 * - Used in signup and plan change flows
 *
 * @component
 * @selector app-subscription-processing
 * @standalone true
 */
@Component({
  selector: 'app-subscription-processing',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './subscription-processing.component.html',
  styleUrl: './subscription-processing.component.scss'
})
export class SubscriptionProcessingComponent implements OnInit, OnDestroy {
  @Input() config: SubscriptionProcessingConfig = {
    paymentId: '',
    userId: '',
    context: 'signup'
  };
  @Output() paymentSuccess = new EventEmitter<void>();
  @Output() paymentError = new EventEmitter<void>();
  @Output() goBack = new EventEmitter<void>();

  processingStatus = 'processing';
  errorMessage = '';
  private statusCheckInterval: any;
  private subscriptionService = inject(SubscriptionService);

  ngOnInit(): void {
    this.startStatusListener();
  }

  ngOnDestroy(): void {
    if (this.statusCheckInterval) {
      clearInterval(this.statusCheckInterval);
    }
  }

  // TODO: IMPLEMENTAR ENDPOINT DE VERIFICACIÓN DE PAGO - Reemplazar polling con API real
  private startStatusListener(): void {
    // Verificar estado del pago cada 2 segundos
    this.statusCheckInterval = setInterval(async () => {
      try {
        const subscription = await
this.subscriptionService.getSubscriptionById(this.config.userId, this.config.paymentId);
        if (subscription) {
          switch (subscription.status) {
            case UserStatus.PURCHASED:
              this.processingStatus = 'success';
              this.paymentSuccess.emit();
              this.clearInterval();
              break;
            case UserStatus.CREATED:
              // Aún procesando
              break;
            default:
              this.processingStatus = 'error';
              this.errorMessage = 'Error processing the subscription';
              this.paymentError.emit();
              this.clearInterval();
              break;
```

```
          }
        } else {
          this.processingStatus = 'error';
          this.errorMessage = 'Subscription not found';
          this.paymentError.emit();
          this.clearInterval();
        }
      } catch (error) {
        console.error('Error verifying subscription status:', error);
        this.processingStatus = 'error';
        this.errorMessage = 'Connection error';
        this.paymentError.emit();
        this.clearInterval();
      }
    }, 2000);

    // Timeout after 30 seconds
    setTimeout(() => {
      if (this.processingStatus === 'processing') {
        this.processingStatus = 'error';
        this.errorMessage = 'Timeout';
        this.paymentError.emit();
        this.clearInterval();
      }
    }, 30000);
  }

  private clearInterval(): void {
    if (this.statusCheckInterval) {
      clearInterval(this.statusCheckInterval);
      this.statusCheckInterval = null;
    }
  }

  onGoBack(): void {
    this.clearInterval();
    this.goBack.emit();
  }

  onRetry(): void {
    this.processingStatus = 'processing';
    this.errorMessage = '';
    this.startStatusListener();
  }

  getContextTitle(): string {
    switch (this.config.context) {
      case 'signup':
        return 'Almost there! Redirecting to finalize your payment.';
      case 'plan-change':
        return 'Almost there! Processing your plan change.';
      default:
        return 'Almost there! Redirecting to finalize your payment.';
    }
  }

  getSuccessMessage(): string {
    switch (this.config.context) {
      case 'signup':
        return 'Your subscription has been activated successfully.';
      case 'plan-change':
        return `Your plan has been successfully changed to ${this.config.planName || 'the
new plan'}.`;
      default:
        return 'Your subscription has been activated successfully.';
    }
  }
}
```

Ø=ÜÁ shared\components\text-input

Ø=ÜÄ shared\components\text-input\text-input.component.ts

```typescript
import { Component, Input, forwardRef } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ControlValueAccessor, FormsModule, ReactiveFormsModule, NG_VALUE_ACCESSOR } from
'@angular/forms';
/**
 * Reusable text input component with Angular Forms integration.
 *
 * This component provides a customizable text input that integrates with
 * Angular Reactive Forms and Template-driven Forms through ControlValueAccessor.
 * It supports labels, placeholders, validation states, and disabled states.
 *
 * Features:
 * - Angular Forms integration (ControlValueAccessor)
 * - Customizable label and placeholder
 * - Support for different input types (text, email, password, etc.)
 * - Required field indicator
 * - Disabled state support
 * - Touch state tracking
 * - Validation error display
 *
 * Usage:
 * <app-text-input
 *   formControlName="email"
 *   label="Email"
 *   placeholder="Enter your email"
 *   type="email"
 *   [required]="true">
 * </app-text-input>
 *
 * Relations:
 * - Used in forms throughout the application
 * - Integrates with Angular Reactive Forms and Template-driven Forms
 *
 * @component
 * @selector app-text-input
 * @standalone true
 */
@Component({
  selector: 'app-text-input',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule],
  templateUrl: './text-input.component.html',
  styleUrls: ['./text-input.component.scss'],
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => TextInputComponent),
      multi: true
    }
  ]
})
export class TextInputComponent implements ControlValueAccessor {
  @Input() label: string = '';
  @Input() placeholder: string = '';
  @Input() type: string = 'text';
  @Input() required: boolean = false;
  @Input() disabled: boolean = false;

  value: string = '';
  touched: boolean = false;

```

```
  onChange = (value: string) => {};
  onTouched = () => {};

  writeValue(value: string): void {
    this.value = value;
  }

  registerOnChange(fn: any): void {
    this.onChange = fn;
  }

  registerOnTouched(fn: any): void {
    this.onTouched = fn;
  }

  setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }

  onInput(event: Event): void {
    const value = (event.target as HTMLInputElement).value;
    this.value = value;
    this.onChange(value);
  }

  onBlur(): void {
    if (!this.touched) {
      this.touched = true;
      this.onTouched();
    }
  }
}

```

Ø=ÜÁ shared\context

Ø=ÜÄ shared\context\constants.ts

```
// Constantes para el contexto de la aplicación

export const CACHE_CONFIG = {
  // TTL en milisegundos
  TRADE_LOCKER_TTL: 5 * 60 * 1000, // 5 minutos
  API_TTL: 10 * 60 * 1000, // 10 minutos
  USER_DATA_TTL: 30 * 60 * 1000, // 30 minutos

  // Tamaños máximos
  MAX_CACHE_SIZE: 100,
  MAX_TRADE_LOCKER_ACCOUNTS: 50,
  MAX_PLUGIN_HISTORY: 1000,

  // Intervalos de actualización
  AUTO_REFRESH_INTERVAL: 60 * 1000, // 1 minuto
  STALE_DATA_THRESHOLD: 2 * 60 * 1000, // 2 minutos
} as const;

export const LOADING_STATES = {
  USER: 'user',
  ACCOUNTS: 'accounts',
  STRATEGIES: 'strategies',
  PLAN: 'plan',
  PLUGIN_HISTORY: 'pluginHistory',
  TRADE_LOCKER: 'tradeLocker',
```

```
} as const;

export const ERROR_TYPES = {
  USER: 'user',
  ACCOUNTS: 'accounts',
  STRATEGIES: 'strategies',
  PLAN: 'plan',
  PLUGIN_HISTORY: 'pluginHistory',
  TRADE_LOCKER: 'tradeLocker',
} as const;

export const CONTEXT_EVENTS = {
  USER_LOGIN: 'user:login',
  USER_LOGOUT: 'user:logout',
  USER_UPDATE: 'user:update',
  ACCOUNTS_ADD: 'accounts:add',
  ACCOUNTS_UPDATE: 'accounts:update',
  ACCOUNTS_REMOVE: 'accounts:remove',
  STRATEGIES_ADD: 'strategies:add',
  STRATEGIES_UPDATE: 'strategies:update',
  STRATEGIES_REMOVE: 'strategies:remove',
  STRATEGIES_ACTIVATE: 'strategies:activate',
  PLAN_UPDATE: 'plan:update',
  PLUGIN_HISTORY_ADD: 'pluginHistory:add',
  CACHE_CLEAR: 'cache:clear',
  ERROR_SET: 'error:set',
  ERROR_CLEAR: 'error:clear',
} as const;

export const DEFAULT_PLAN = {
  planId: 'free',
  planName: 'Free',
  maxAccounts: 1,
  maxStrategies: 1,
  features: ['basic_trading'],
  isActive: true,
} as const;

export const PLUGIN_ACTIONS = {
  ACCOUNT_CREATED: 'account_created',
  ACCOUNT_UPDATED: 'account_updated',
  ACCOUNT_DELETED: 'account_deleted',
  STRATEGY_CREATED: 'strategy_created',
  STRATEGY_UPDATED: 'strategy_updated',
  STRATEGY_DELETED: 'strategy_deleted',
  STRATEGY_ACTIVATED: 'strategy_activated',
  STRATEGY_DEACTIVATED: 'strategy_deactivated',
  PLAN_CHANGED: 'plan_changed',
  USER_LOGIN: 'user_login',
  USER_LOGOUT: 'user_logout',
} as const;

```

Ø=ÜÄ shared\context\context.ts

```
import { Injectable, signal, computed, effect } from '@angular/core';
import { BehaviorSubject, Observable, combineLatest, map, distinctUntilChanged } from 'rxjs';
import { User } from '../../features/overview/models/overview';
import { AccountData } from '../../features/auth/models/userModel';
import { ConfigurationOverview } from '../../features/strategy/models/strategy.model';
import { BalanceData, GroupedTradeFinal } from '../../features/report/models/report.model';
import { Plan } from '../services/planService';
import { TradeLockerApiService } from '../services/tradelocker-api.service';

// Interfaces para datos de API externa
export interface TradeLockerAccountData {
```

```
  accountId: string;
  balance: BalanceData;
  lastUpdated: number;
  isValid: boolean;
}

export interface PluginHistoryData {
  id: string;
  userId: string;
  action: string;
  timestamp: number;
  details: any;
}

export interface UserPlanData {
  planId: string;
  planName: string;
  maxAccounts: number;
  maxStrategies: number;
  features: string[];
  isActive: boolean;
  expiresAt?: number;
}

export interface AppContextState {
  // Usuario autenticado
  currentUser: User | null;
  isAuthenticated: boolean;

  // Datos del usuario
  userAccounts: AccountData[];
  userStrategies: ConfigurationOverview[];
  userPlan: UserPlanData | null;
  pluginHistory: PluginHistoryData[];

  // Planes globales (cargados una vez al login)
  globalPlans: Plan[];
  planLimits: { [planName: string]: { tradingAccounts: number; strategies: number } };

  // Datos de reportes
  reportData: {
    accountHistory: any[];
    stats: any;
    balanceData: any;
    monthlyReports: any[];
  };

  // Trading history por cuenta (nuevo)
  tradingHistoryByAccount: Map<string, {
    accountHistory: GroupedTradeFinal[];
    stats: any;
    balanceData: any;
    lastUpdated: number;
  }>;

  // Datos de overview (para admins)
  overviewData: {
    allUsers: User[];
    subscriptions: any[];
    monthlyReports: any[];
    allAccounts: AccountData[];
    allStrategies: ConfigurationOverview[];
  };

  // Datos de API externa (con caché)
  tradeLockerData: Map<string, TradeLockerAccountData>;
  apiCache: Map<string, { data: any; timestamp: number; ttl: number }>;

  // Estados de carga
  isLoading: {
```

```
    user: boolean;
    accounts: boolean;
    strategies: boolean;
    plan: boolean;
    pluginHistory: boolean;
    tradeLocker: boolean;
    report: boolean;
    overview: boolean;
    globalPlans: boolean;
  };

  // Estados de error
  errors: {
    user: string | null;
    accounts: string | null;
    strategies: string | null;
    plan: string | null;
    pluginHistory: string | null;
    tradeLocker: string | null;
    report: string | null;
    overview: string | null;
    globalPlans: string | null;
  };

  // Configuración de caché
  cacheConfig: {
    tradeLockerTtl: number; // 5 minutos
    apiTtl: number; // 10 minutos
    maxCacheSize: number; // 100 elementos
  };
}

/**
 * Global application context service for managing shared state.
 *
 * This service provides a centralized state management system using Angular signals
 * and RxJS observables. It manages user data, accounts, strategies, plans, trading
 * history, and API caching across the entire application.
 *
 * Features:
 * - User state management (current user, authentication status)
 * - Account management (CRUD operations)
 * - Strategy management (CRUD operations, activation)
 * - Plan management (user plan, global plans, limits)
 * - Trading history per account (with localStorage persistence)
 * - API caching (TradeLocker data, general API cache)
 * - Report data management
 * - Overview data management (for admins)
 * - Loading states per component
 * - Error states per component
 * - Computed signals for derived data
 *
 * State Management:
 * - Signals: For reactive UI updates (Angular signals)
 * - Observables: For RxJS-based subscriptions
 * - BehaviorSubject: Internal state management
 * - localStorage: Persistence for trading history
 *
 * Caching:
 * - TradeLocker data: 5-minute TTL
 * - General API cache: 10-minute TTL
 * - Automatic cache eviction (20% oldest entries when max size reached)
 *
 * Relations:
 * - TradeLockerApiService: Fetches trading data
 * - All feature modules: Consume and update context data
 * - localStorage: Persists trading history
 *
 * @service
 * @injectable
```

```
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class AppContextService {
  // Estado inicial
  private initialState: AppContextState = {
    currentUser: null,
    isAuthenticated: false,
    userAccounts: [],
    userStrategies: [],
    userPlan: null,
    pluginHistory: [],
    globalPlans: [],
    planLimits: {},
    reportData: {
      accountHistory: [],
      stats: null,
      balanceData: null,
      monthlyReports: []
    },
    overviewData: {
      allUsers: [],
      subscriptions: [],
      monthlyReports: [],
      allAccounts: [],
      allStrategies: []
    },
    tradeLockerData: new Map(),
    apiCache: new Map(),
    tradingHistoryByAccount: new Map(),
    isLoading: {
      user: false,
      accounts: false,
      strategies: false,
      plan: false,
      pluginHistory: false,
      tradeLocker: false,
      report: false,
      overview: false,
      globalPlans: false
    },
    errors: {
      user: null,
      accounts: null,
      strategies: null,
      plan: null,
      pluginHistory: null,
      tradeLocker: null,
      report: null,
      overview: null,
      globalPlans: null
    },
    cacheConfig: {
      tradeLockerTtl: 5 * 60 * 1000, // 5 minutos
      apiTtl: 10 * 60 * 1000, // 10 minutos
      maxCacheSize: 100
    }
  };

  // Estado reactivo principal
  private stateSubject = new BehaviorSubject<AppContextState>(this.initialState);
  public state$ = this.stateSubject.asObservable();

  // Signals para datos específicos (más eficientes para UI)
  public currentUser = signal<User | null>(null);
  public isAuthenticated = signal<boolean>(false);
  public userAccounts = signal<AccountData[]>([]);
  public userStrategies = signal<ConfigurationOverview[]>([]);
```

```
  public userPlan = signal<UserPlanData | null>(null);
  public pluginHistory = signal<PluginHistoryData[]>([]);

  // Signals para planes globales
  public globalPlans = signal<Plan[]>([]);
  public planLimits = signal<{ [planName: string]: { tradingAccounts: number; strategies:
number } }>({});
  // Signals para datos de reportes
  public reportData = signal<{
    accountHistory: any[];
    stats: any;
    balanceData: any;
    monthlyReports: any[];
  }>({
    accountHistory: [],
    stats: null,
    balanceData: null,
    monthlyReports: []
  });

  // Signals para datos de overview
  public overviewData = signal<{
    allUsers: User[];
    subscriptions: any[];
    monthlyReports: any[];
    allAccounts: AccountData[];
    allStrategies: ConfigurationOverview[];
  }>({
    allUsers: [],
    subscriptions: [],
    monthlyReports: [],
    allAccounts: [],
    allStrategies: []
  });

  // Computed signals para datos derivados
  public activeStrategies = computed(() =>
    this.userStrategies().filter(strategy => strategy.status === true)
  );

  public totalAccounts = computed(() =>
    this.userAccounts().length
  );

  public totalStrategies = computed(() =>
    this.userStrategies().length
  );

  public canCreateAccount = computed(() => {
    const plan = this.userPlan();
    const currentCount = this.totalAccounts();
    return plan ? currentCount < plan.maxAccounts : false;
  });

  public canCreateStrategy = computed(() => {
    const plan = this.userPlan();
    const currentCount = this.totalStrategies();
    return plan ? currentCount < plan.maxStrategies : false;
  });

  public planLimitations = computed(() => ({
    maxAccounts: this.userPlan()?.maxAccounts || 0,
    currentAccounts: this.totalAccounts(),
    maxStrategies: this.userPlan()?.maxStrategies || 0,
    currentStrategies: this.totalStrategies(),
    canCreateAccount: this.canCreateAccount(),
    canCreateStrategy: this.canCreateStrategy()
  }));

  // Computed signals para planes globales
```

```
  public orderedPlans = computed(() => {
    const plans = this.globalPlans();

    const orderedPlanNames = ['Free', 'Starter', 'Pro'];
    const orderedPlans: Plan[] = [];

    orderedPlanNames.forEach(planName => {
      const plan = plans.find(p => p.name.toLowerCase() === planName.toLowerCase());
      if (plan) {
        orderedPlans.push(plan);
      }
    });

    return orderedPlans;
  });

  // Observables específicos para suscripciones
  public currentUser$ = this.state$.pipe(
    map(state => state.currentUser),
    distinctUntilChanged()
  );

  public userAccounts$ = this.state$.pipe(
    map(state => state.userAccounts),
    distinctUntilChanged()
  );

  public userStrategies$ = this.state$.pipe(
    map(state => state.userStrategies),
    distinctUntilChanged()
  );

  public userPlan$ = this.state$.pipe(
    map(state => state.userPlan),
    distinctUntilChanged()
  );

  public pluginHistory$ = this.state$.pipe(
    map(state => state.pluginHistory),
    distinctUntilChanged()
  );

  public isLoading$ = this.state$.pipe(
    map(state => state.isLoading),
    distinctUntilChanged()
  );

  public errors$ = this.state$.pipe(
    map(state => state.errors),
    distinctUntilChanged()
  );

  public reportData$ = this.state$.pipe(
    map(state => state.reportData),
    distinctUntilChanged()
  );

  public overviewData$ = this.state$.pipe(
    map(state => state.overviewData),
    distinctUntilChanged()
  );

  public globalPlans$ = this.state$.pipe(
    map(state => state.globalPlans),
    distinctUntilChanged()
  );

  public planLimits$ = this.state$.pipe(
    map(state => state.planLimits),
    distinctUntilChanged()
```

```
  );

  constructor(private tradeLockerApi: TradeLockerApiService) {
    // Sincronizar signals con el estado principal
    effect(() => {
      this.updateState({
        currentUser: this.currentUser(),
        isAuthenticated: this.isAuthenticated(),
        userAccounts: this.userAccounts(),
        userStrategies: this.userStrategies(),
        userPlan: this.userPlan(),
        pluginHistory: this.pluginHistory(),
        globalPlans: this.globalPlans(),
        planLimits: this.planLimits(),
        reportData: this.reportData(),
        overviewData: this.overviewData()
      });
    });
  }

  // ===== MÉTODOS DE ACTUALIZACIÓN DE ESTADO =====

  private updateState(updates: Partial<AppContextState>): void {
    const currentState = this.stateSubject.value;
    const newState = { ...currentState, ...updates };
    this.stateSubject.next(newState);
  }

  // ===== MÉTODOS DE USUARIO =====

  setCurrentUser(user: User | null): void {
    this.currentUser.set(user);
    this.isAuthenticated.set(user !== null);

    if (user) {
      this.updateState({
        currentUser: user,
        isAuthenticated: true
      });

      // NO cargar trading history automáticamente para evitar peticiones repetidas
      // this.loadTradingHistoryAfterLogin(user.id);
    } else {
      this.clearUserData();
    }
  }

  updateUserData(userData: Partial<User>): void {
    const currentUser = this.currentUser();
    if (currentUser) {
      const updatedUser = { ...currentUser, ...userData };
      this.currentUser.set(updatedUser);
    }
  }

  clearUserData(): void {
    this.currentUser.set(null);
    this.isAuthenticated.set(false);
    this.userAccounts.set([]);
    this.userStrategies.set([]);
    this.userPlan.set(null);
    this.pluginHistory.set([]);
    // No limpiar globalPlans y planLimits ya que son globales

    this.updateState({
      currentUser: null,
      isAuthenticated: false,
      userAccounts: [],
      userStrategies: [],
      userPlan: null,
```

```
      pluginHistory: []
    });
  }

  // ===== MÉTODOS DE CUENTAS =====

  setUserAccounts(accounts: AccountData[]): void {
    this.userAccounts.set(accounts);
  }

  addAccount(account: AccountData): void {
    const currentAccounts = this.userAccounts();
    this.userAccounts.set([...currentAccounts, account]);
  }

  updateAccount(accountId: string, updates: Partial<AccountData>): void {
    const currentAccounts = this.userAccounts();
    const updatedAccounts = currentAccounts.map(account =>
      account.id === accountId ? { ...account, ...updates } : account
    );
    this.userAccounts.set(updatedAccounts);
  }

  removeAccount(accountId: string): void {
    const currentAccounts = this.userAccounts();
    const filteredAccounts = currentAccounts.filter(account => account.id !== accountId);
    this.userAccounts.set(filteredAccounts);
  }

  // ===== MÉTODOS DE ESTRATEGIAS =====

  setUserStrategies(strategies: ConfigurationOverview[]): void {
    this.userStrategies.set(strategies);
  }

  addStrategy(strategy: ConfigurationOverview & { id: string }): void {
    const currentStrategies = this.userStrategies();
    this.userStrategies.set([...currentStrategies, strategy]);
  }

  updateStrategy(strategyId: string, updates: Partial<ConfigurationOverview>): void {
    const currentStrategies = this.userStrategies();
    const updatedStrategies = currentStrategies.map(strategy =>
      (strategy as any).id === strategyId ? { ...strategy, ...updates } : strategy
    );
    this.userStrategies.set(updatedStrategies);
  }

  removeStrategy(strategyId: string): void {
    const currentStrategies = this.userStrategies();
    const filteredStrategies = currentStrategies.filter(strategy => (strategy as any).id !==
strategyId);
    this.userStrategies.set(filteredStrategies);
  }

  activateStrategy(strategyId: string): void {
    const currentStrategies = this.userStrategies();
    const updatedStrategies = currentStrategies.map(strategy => ({
      ...strategy,
      status: (strategy as any).id === strategyId
    }));
    this.userStrategies.set(updatedStrategies);
  }

  // ===== MÉTODOS DE PLAN =====

  setUserPlan(plan: UserPlanData | null): void {
    this.userPlan.set(plan);
  }

  updateUserPlan(planUpdates: Partial<UserPlanData>): void {
```

```
    const currentPlan = this.userPlan();
    if (currentPlan) {
      const updatedPlan = { ...currentPlan, ...planUpdates };
      this.userPlan.set(updatedPlan);
    }
  }

  // ===== MÉTODOS DE PLANES GLOBALES =====

  setGlobalPlans(plans: Plan[]): void {
    this.globalPlans.set(plans);

    // Crear mapa de límites automáticamente
    const limits: { [planName: string]: { tradingAccounts: number; strategies: number } } =
{}; plans.forEach(plan => {
      // Usar el nombre original del plan como clave
      limits[plan.name] = {
        tradingAccounts: plan.tradingAccounts,
        strategies: plan.strategies
      };
    });
    this.planLimits.set(limits);
  }

  getPlanByName(planName: string): Plan | undefined {
    const plans = this.globalPlans();
    return plans.find(plan => plan.name.toLowerCase() === planName.toLowerCase());
  }

  getPlanById(planId: string): Plan | undefined {
    const plans = this.globalPlans();
    return plans.find(plan => plan.id === planId);
  }

  getPlanLimits(planName: string): { tradingAccounts: number; strategies: number } | null {
    const limits = this.planLimits();
    // Buscar por nombre exacto primero
    if (limits[planName]) {
      return limits[planName];
    }

    // Si no se encuentra, buscar case-insensitive
    const planKey = Object.keys(limits).find(key =>
      key.toLowerCase() === planName.toLowerCase()
    );

    return planKey ? limits[planKey] : null;
  }

  // ===== MÉTODOS DE HISTORIAL DE PLUGINS =====

  setPluginHistory(history: PluginHistoryData[]): void {
    this.pluginHistory.set(history);
  }

  addPluginHistoryEntry(entry: PluginHistoryData): void {
    const currentHistory = this.pluginHistory();
    this.pluginHistory.set([entry, ...currentHistory]);
  }

  // ===== MÉTODOS DE CACHÉ DE API =====

  setTradeLockerData(accountId: string, data: TradeLockerAccountData): void {
    const currentState = this.stateSubject.value;
    const newTradeLockerData = new Map(currentState.tradeLockerData);
    newTradeLockerData.set(accountId, data);

    this.updateState({
      tradeLockerData: newTradeLockerData
    });
```

```
  }

  getTradeLockerData(accountId: string): TradeLockerAccountData | null {
    const currentState = this.stateSubject.value;
    return currentState.tradeLockerData.get(accountId) || null;
  }

  isTradeLockerDataValid(accountId: string): boolean {
    const data = this.getTradeLockerData(accountId);
    if (!data) return false;

    const now = Date.now();
    const ttl = this.stateSubject.value.cacheConfig.tradeLockerTtl;
    return (now - data.lastUpdated) < ttl;
  }

  setApiCache(key: string, data: any, ttl?: number): void {
    const currentState = this.stateSubject.value;
    const newApiCache = new Map(currentState.apiCache);
    const cacheTtl = ttl || currentState.cacheConfig.apiTtl;

    newApiCache.set(key, {
      data,
      timestamp: Date.now(),
      ttl: cacheTtl
    });

    // Limpiar caché si excede el tamaño máximo
    if (newApiCache.size > currentState.cacheConfig.maxCacheSize) {
      const entries = Array.from(newApiCache.entries());
      entries.sort((a, b) => a[1].timestamp - b[1].timestamp);

      // Eliminar el 20% más antiguo
      const toRemove = Math.floor(entries.length * 0.2);
      for (let i = 0; i < toRemove; i++) {
        newApiCache.delete(entries[i][0]);
      }
    }

    this.updateState({
      apiCache: newApiCache
    });
  }

  getApiCache(key: string): any | null {
    const currentState = this.stateSubject.value;
    const cached = currentState.apiCache.get(key);

    if (!cached) return null;

    const now = Date.now();
    if ((now - cached.timestamp) > cached.ttl) {
      // Expirar caché
      const newApiCache = new Map(currentState.apiCache);
      newApiCache.delete(key);
      this.updateState({ apiCache: newApiCache });
      return null;
    }

    return cached.data;
  }

  clearApiCache(): void {
    this.updateState({
      tradeLockerData: new Map(),
      apiCache: new Map()
    });
  }

  // ===== MÉTODOS DE ESTADO DE CARGA =====
```

```
642
643   setLoading(component: keyof AppContextState['isLoading'], loading: boolean): void {
644     const currentState = this.stateSubject.value;
645     this.updateState({
646       isLoading: {
647         ...currentState.isLoading,
648         [component]: loading
649       }
650     });
651   }
652
653   setError(component: keyof AppContextState['errors'], error: string | null): void {
654     const currentState = this.stateSubject.value;
655     this.updateState({
656       errors: {
657         ...currentState.errors,
658         [component]: error
659       }
660     });
661   }
662
663   clearErrors(): void {
664     this.updateState({
665       errors: {
666         user: null,
667         accounts: null,
668         strategies: null,
669         plan: null,
670         pluginHistory: null,
671         tradeLocker: null,
672         report: null,
673         overview: null,
674         globalPlans: null
675       }
676     });
677   }
678
679   // ===== MÉTODOS DE UTILIDAD =====
680
681   getCurrentState(): AppContextState {
682     return this.stateSubject.value;
683   }
684
685   resetState(): void {
686     this.stateSubject.next(this.initialState);
687     this.currentUser.set(null);
688     this.isAuthenticated.set(false);
689     this.userAccounts.set([]);
690     this.userStrategies.set([]);
691     this.userPlan.set(null);
692     this.pluginHistory.set([]);
693     this.globalPlans.set([]);
694     this.planLimits.set({});
695   }
696
697   // ===== MÉTODOS DE VALIDACIÓN =====
698
699   isDataStale(component: keyof AppContextState['isLoading'], maxAge: number = 5 * 60 *
700 1000): boolean {
701     // Implementar lógica para verificar si los datos están obsoletos
702     // Esto se puede usar para decidir si hacer nuevas peticiones
703     return true; // Placeholder
704   }
705
706   // ===== MÉTODOS DE SUSCRIPCIÓN =====
707
708   subscribeToUserChanges(): Observable<User | null> {
709     return this.currentUser$;
710   }
711
712   subscribeToAccountsChanges(): Observable<AccountData[]> {
```

```
    return this.userAccounts$;
  }

  subscribeToStrategiesChanges(): Observable<ConfigurationOverview[]> {
    return this.userStrategies$;
  }

  subscribeToPlanChanges(): Observable<UserPlanData | null> {
    return this.userPlan$;
  }

  subscribeToPluginHistoryChanges(): Observable<PluginHistoryData[]> {
    return this.pluginHistory$;
  }

  subscribeToGlobalPlansChanges(): Observable<Plan[]> {
    return this.globalPlans$;
  }

  subscribeToPlanLimitsChanges(): Observable<{ [planName: string]: { tradingAccounts:
number; strategies: number } }> {
    return this.planLimits$;
  }

  // ===== MÉTODOS DE DATOS DE REPORTES =====

  setReportData(data: {
    accountHistory?: any[];
    stats?: any;
    balanceData?: any;
    monthlyReports?: any[];
  }): void {
    const currentData = this.reportData();
    this.reportData.set({ ...currentData, ...data });
  }

  updateReportStats(stats: any): void {
    const currentData = this.reportData();
    this.reportData.set({ ...currentData, stats });
  }

  updateReportBalance(balanceData: any): void {
    const currentData = this.reportData();
    this.reportData.set({ ...currentData, balanceData });
  }

  updateReportHistory(accountHistory: any[]): void {
    const currentData = this.reportData();
    this.reportData.set({ ...currentData, accountHistory });
  }

  // ===== MÉTODOS DE DATOS DE OVERVIEW =====

  setOverviewData(data: {
    allUsers?: User[];
    subscriptions?: any[];
    monthlyReports?: any[];
    allAccounts?: AccountData[];
    allStrategies?: ConfigurationOverview[];
  }): void {
    const currentData = this.overviewData();
    this.overviewData.set({ ...currentData, ...data });
  }

  updateOverviewUsers(allUsers: User[]): void {
    const currentData = this.overviewData();
    this.overviewData.set({ ...currentData, allUsers });
  }

  updateOverviewSubscriptions(subscriptions: any[]): void {
    const currentData = this.overviewData();
```

```
    this.overviewData.set({ ...currentData, subscriptions });
  }

  updateOverviewAccounts(allAccounts: AccountData[]): void {
    const currentData = this.overviewData();
    this.overviewData.set({ ...currentData, allAccounts });
  }

  updateOverviewStrategies(allStrategies: ConfigurationOverview[]): void {
    const currentData = this.overviewData();
    this.overviewData.set({ ...currentData, allStrategies });
  }

  // ===== MÉTODOS DE TRADING HISTORY POR CUENTA =====

  /**
   * Cargar trading history de la primera cuenta después del login
   */
  private async loadTradingHistoryAfterLogin(userId: string): Promise<void> {
    try {
      // Esperar a que las cuentas estén cargadas
      const accounts = this.userAccounts();
      if (accounts.length === 0) {
        // Si no hay cuentas, intentar cargarlas
        await this.loadUserAccountsIfNeeded(userId);
        return;
      }

      // Cargar trading history de la primera cuenta
      const firstAccount = accounts[0];
      await this.loadTradingHistoryForAccount(firstAccount);
    } catch (error) {
      console.error('Error loading trading history after login:', error);
    }
  }

  /**
   * Cargar cuentas del usuario si no están cargadas
   */
  private async loadUserAccountsIfNeeded(userId: string): Promise<void> {
    // Este método se implementará cuando se integre con AuthService
    // Por ahora, solo logueamos que necesitamos cargar las cuentas
    console.log('Loading user accounts for userId:', userId);
  }

  /**
   * Cargar trading history para una cuenta específica
   */
  async loadTradingHistoryForAccount(account: AccountData): Promise<void> {
    try {
      this.setLoading('tradeLocker', true);
      this.setError('tradeLocker', null);

      // Obtener userKey
      const userKey = await this.tradeLockerApi.getUserKey(
        account.emailTradingAccount,
        account.brokerPassword,
        account.server
      ).toPromise();

      if (!userKey) {
        throw new Error('No se pudo obtener userKey');
      }

      // Obtener balance data
      const balanceData = await this.tradeLockerApi.getAccountBalance(
        account.accountID,
        userKey,
        1
      ).toPromise();
```

```
852
853       // Obtener trading history
854       const tradingHistory = await this.tradeLockerApi.getTradingHistory(
855         userKey,
856         account.accountID,
857         1
858       ).toPromise();
859
860       // Calcular estadísticas
861       const stats = this.calculateStatsFromTrades(tradingHistory || []);
862
863       // Guardar datos por cuenta
864       const accountData = {
865         accountHistory: tradingHistory || [],
866         stats: stats,
867         balanceData: balanceData,
868         lastUpdated: Date.now()
869       };
870
871       this.setTradingHistoryForAccount(account.id, accountData);
872
873       // Guardar en localStorage
874       this.saveTradingHistoryToLocalStorage(account.id, accountData);
875
876       this.setLoading('tradeLocker', false);
877     } catch (error) {
878       console.error('Error loading trading history for account:', error);
879       this.setLoading('tradeLocker', false);
880       this.setError('tradeLocker', 'Error al cargar trading history');
881     }
882   }
883
884   /**
885    * Calcular estadísticas desde los trades
886    */
887   private calculateStatsFromTrades(trades: GroupedTradeFinal[]): any {
888     if (!trades || trades.length === 0) {
889       return {
890         netPnl: 0,
891         tradeWinPercent: 0,
892         profitFactor: 0,
893         avgWinLossTrades: 0,
894         totalTrades: 0,
895         activePositions: 0
896       };
897     }
898
899     const normalizedTrades = trades.map(trade => ({
900       ...trade,
901       pnl: trade.pnl ?? 0
902     }));
903
904     const totalGains = normalizedTrades
905       .filter(t => t.pnl > 0)
906       .reduce((sum, t) => sum + t.pnl, 0);
907
908     const totalLosses = Math.abs(normalizedTrades
909       .filter(t => t.pnl < 0)
910       .reduce((sum, t) => sum + t.pnl, 0));
911
912     const winningTrades = normalizedTrades.filter(t => t.pnl > 0).length;
913     const totalTrades = normalizedTrades.length;
914     const winPercent = totalTrades > 0 ? (winningTrades / totalTrades) * 100 : 0;
915     const profitFactor = totalLosses > 0 ? totalGains / totalLosses : (totalGains > 0 ?
916 999.99 : 0);
917     return {
918       netPnl: Math.round((totalGains - totalLosses) * 100) / 100,
919       tradeWinPercent: Math.round(winPercent * 100) / 100,
920       profitFactor: Math.round(profitFactor * 100) / 100,
921       avgWinLossTrades: 0, // Se calculará si es necesario
```

```
      totalTrades: totalTrades,
      activePositions: trades.filter(trade => trade.isOpen === true).length
    };
  }

  /**
   * Establecer trading history para una cuenta específica
   */
  setTradingHistoryForAccount(accountId: string, data: {
    accountHistory: GroupedTradeFinal[];
    stats: any;
    balanceData: any;
    lastUpdated: number;
  }): void {
    const currentState = this.stateSubject.value;
    const newTradingHistory = new Map(currentState.tradingHistoryByAccount);
    newTradingHistory.set(accountId, data);

    this.updateState({
      tradingHistoryByAccount: newTradingHistory
    });
  }

  /**
   * Obtener trading history para una cuenta específica
   */
  getTradingHistoryForAccount(accountId: string): {
    accountHistory: GroupedTradeFinal[];
    stats: any;
    balanceData: any;
    lastUpdated: number;
  } | null {
    const currentState = this.stateSubject.value;
    return currentState.tradingHistoryByAccount.get(accountId) || null;
  }

  /**
   * Guardar trading history en localStorage por cuenta
   */
  saveTradingHistoryToLocalStorage(accountId: string, data: {
    accountHistory: GroupedTradeFinal[];
    stats: any;
    balanceData: any;
    lastUpdated: number;
  }): void {
    try {
      const key = `tradeSwitch_tradingHistory_${accountId}`;

      // Cargar datos existentes para preservar balanceData si es null
      let existingBalanceData = null;
      try {
        const existingData = localStorage.getItem(key);
        if (existingData) {
          const parsed = JSON.parse(existingData);
          existingBalanceData = parsed.balanceData;
        }
      } catch (error) {
        // Si no hay datos existentes, continuar
      }

      const dataToSave = {
        accountHistory: data.accountHistory,
        stats: data.stats,
        // Solo sobrescribir balanceData si no es null, de lo contrario preservar el existente
        balanceData: data.balanceData !== null && data.balanceData !== undefined
          ? data.balanceData
          : existingBalanceData,
        lastUpdated: data.lastUpdated
      };
```

```
      localStorage.setItem(key, JSON.stringify(dataToSave));
    } catch (error) {
      console.error('Error saving trading history to localStorage:', error);
    }
  }

  /**
   * Cargar trading history desde localStorage por cuenta
   */
  loadTradingHistoryFromLocalStorage(accountId: string): {
    accountHistory: GroupedTradeFinal[];
    stats: any;
    balanceData: any;
    lastUpdated: number;
  } | null {
    try {
      const key = `tradeSwitch_tradingHistory_${accountId}`;
      const data = localStorage.getItem(key);
      if (data) {
        return JSON.parse(data);
      }
    } catch (error) {
      console.error('Error loading trading history from localStorage:', error);
    }
    return null;
  }

  /**
   * Limpiar trading history de una cuenta específica
   */
  clearTradingHistoryForAccount(accountId: string): void {
    const currentState = this.stateSubject.value;
    const newTradingHistory = new Map(currentState.tradingHistoryByAccount);
    newTradingHistory.delete(accountId);

    this.updateState({
      tradingHistoryByAccount: newTradingHistory
    });

    // Limpiar también del localStorage
    try {
      const key = `tradeSwitch_tradingHistory_${accountId}`;
      localStorage.removeItem(key);
    } catch (error) {
      console.error('Error clearing trading history from localStorage:', error);
    }
  }

  /**
   * Cargar datos de reporte desde localStorage por accountID
   */
  loadReportDataFromLocalStorage(accountID: string): {
    tradingAccount: AccountData;
    accountHistory: GroupedTradeFinal[];
    stats: any;
    balanceData: any;
    lastUpdated: number;
  } | null {
    try {
      const key = `tradeSwitch_reportData_${accountID}`;
      const data = localStorage.getItem(key);
      if (data) {
        return JSON.parse(data);
      }
    } catch (error) {
      console.error('Error loading report data from localStorage:', error);
    }
    return null;
  }

```

```
  /**
   * Guardar datos de reporte en localStorage por accountID
   */
  saveReportDataToLocalStorage(accountID: string, tradingAccount: AccountData, reportData: {
    accountHistory: GroupedTradeFinal[];
    stats: any;
    balanceData: any;
    lastUpdated: number;
  }): void {
    try {
      const key = `tradeSwitch_reportData_${accountID}`;

      // Cargar datos existentes para preservar balanceData si es null
      let existingBalanceData = null;
      try {
        const existingData = localStorage.getItem(key);
        if (existingData) {
          const parsed = JSON.parse(existingData);
          existingBalanceData = parsed.balanceData;
        }
      } catch (error) {
        // Si no hay datos existentes, continuar
      }

      const dataToSave = {
        tradingAccount: tradingAccount,
        accountHistory: reportData.accountHistory,
        stats: reportData.stats,
        // Solo sobrescribir balanceData si no es null, de lo contrario preservar el existente
        balanceData: reportData.balanceData !== null && reportData.balanceData !== undefined
          ? reportData.balanceData
          : existingBalanceData,
        lastUpdated: reportData.lastUpdated
      };

      localStorage.setItem(key, JSON.stringify(dataToSave));
    } catch (error) {
      console.error('Error saving report data to localStorage:', error);
    }
  }

  /**
   * Limpiar datos de reporte de localStorage por accountID
   */
  clearReportDataFromLocalStorage(accountID: string): void {
    try {
      const key = `tradeSwitch_reportData_${accountID}`;
      localStorage.removeItem(key);
    } catch (error) {
      console.error('Error clearing report data from localStorage:', error);
    }
  }
}
```

Ø=ÜÄ shared\context\index.ts

```
// Exportaciones del contexto de la aplicación

export { AppContextService } from './context';
export * from './types';
export * from './constants';
export * from './utils';

// Re-exportar tipos importantes para facilitar el uso
export type {
```

```
  ConfigurationOverviewWithId,
  AccountDataWithId,
  LoadingState,
  ErrorState,
  CacheConfig,
  TradeLockerAccountData,
  PluginHistoryData,
  UserPlanData,
  PlanLimitations,
  CachedData,
  ContextEvent,
  ContextEventListener
} from './types';

```

Ø=ÜÄ shared\context\types.ts

```
// Tipos extendidos para el contexto de la aplicación

import { AccountData } from "../../features/auth/models/userModel";
import { BalanceData } from "../../features/report/models/report.model";
import { ConfigurationOverview } from "../../features/strategy/models/strategy.model";

export interface ConfigurationOverviewWithId extends ConfigurationOverview {
  id: string;
}

export interface AccountDataWithId extends AccountData {
  id: string;
}

// Tipos para estados de carga
export interface LoadingState {
  user: boolean;
  accounts: boolean;
  strategies: boolean;
  plan: boolean;
  pluginHistory: boolean;
  tradeLocker: boolean;
}

// Tipos para estados de error
export interface ErrorState {
  user: string | null;
  accounts: string | null;
  strategies: string | null;
  plan: string | null;
  pluginHistory: string | null;
  tradeLocker: string | null;
}

// Tipos para configuración de caché
export interface CacheConfig {
  tradeLockerTtl: number;
  apiTtl: number;
  maxCacheSize: number;
}

// Tipos para datos de API externa
export interface TradeLockerAccountData {
  accountId: string;
  balance: BalanceData;
  lastUpdated: number;
  isValid: boolean;
}

```

```
export interface PluginHistoryData {
  id: string;
  userId: string;
  action: string;
  timestamp: number;
  details: any;
}

export interface UserPlanData {
  planId: string;
  planName: string;
  maxAccounts: number;
  maxStrategies: number;
  features: string[];
  isActive: boolean;
  expiresAt?: number;
}

// Tipos para limitaciones de plan
export interface PlanLimitations {
  maxAccounts: number;
  currentAccounts: number;
  maxStrategies: number;
  currentStrategies: number;
  canCreateAccount: boolean;
  canCreateStrategy: boolean;
}

// Tipos para datos de caché de API
export interface CachedData<T = any> {
  data: T;
  timestamp: number;
  ttl: number;
}

// Tipos para eventos del contexto
export type ContextEvent =
  | 'user:login'
  | 'user:logout'
  | 'user:update'
  | 'accounts:add'
  | 'accounts:update'
  | 'accounts:remove'
  | 'strategies:add'
  | 'strategies:update'
  | 'strategies:remove'
  | 'strategies:activate'
  | 'plan:update'
  | 'pluginHistory:add'
  | 'cache:clear'
  | 'error:set'
  | 'error:clear';

// Tipos para listeners de eventos
export type ContextEventListener = (event: ContextEvent, data?: any) => void;

```

Ø=ÜÄ shared\context\utils.ts

```
import { CACHE_CONFIG, PLUGIN_ACTIONS } from './constants';
import { TradeLockerAccountData, PluginHistoryData, UserPlanData } from './types';

// Utilidades para el contexto de la aplicación

/**
 * Verifica si los datos están obsoletos basándose en su timestamp
```

```
 */
export function isDataStale(timestamp: number, ttl: number = CACHE_CONFIG.API_TTL): boolean {
  const now = Date.now();
  return (now - timestamp) > ttl;
}

/**
 * Verifica si los datos de TradeLocker están obsoletos
 */
export function isTradeLockerDataStale(data: TradeLockerAccountData): boolean {
  return isDataStale(data.lastUpdated, CACHE_CONFIG.TRADE_LOCKER_TTL);
}

/**
 * Crea una entrada de historial de plugin
 */
export function createPluginHistoryEntry(
  userId: string,
  action: string,
  details: any = {}
): PluginHistoryData {
  return {
    id: generateId(),
    userId,
    action,
    timestamp: Date.now(),
    details
  };
}

/**
 * Genera un ID único
 */
export function generateId(): string {
  return `${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
}

/**
 * Valida si un plan permite crear más cuentas
 */
export function canCreateAccount(plan: UserPlanData | null, currentCount: number): boolean {
  if (!plan) return false;
  return currentCount < plan.maxAccounts;
}

/**
 * Valida si un plan permite crear más estrategias
 */
export function canCreateStrategy(plan: UserPlanData | null, currentCount: number): boolean {
  if (!plan) return false;
  return currentCount < plan.maxStrategies;
}

/**
 * Calcula las limitaciones del plan
 */
export function calculatePlanLimitations(
  plan: UserPlanData | null,
  currentAccounts: number,
  currentStrategies: number
) {
  return {
    maxAccounts: plan?.maxAccounts || 0,
    currentAccounts,
    maxStrategies: plan?.maxStrategies || 0,
    currentStrategies,
    canCreateAccount: canCreateAccount(plan, currentAccounts),
    canCreateStrategy: canCreateStrategy(plan, currentStrategies),
    accountsRemaining: Math.max(0, (plan?.maxAccounts || 0) - currentAccounts),
    strategiesRemaining: Math.max(0, (plan?.maxStrategies || 0) - currentStrategies)
```

```typescript
  };
}

/**
 * Limpia datos obsoletos del caché
 */
export function cleanStaleCache<T>(
  cache: Map<string, { data: T; timestamp: number; ttl: number }>
): Map<string, { data: T; timestamp: number; ttl: number }> {
  const now = Date.now();
  const cleanedCache = new Map();

  for (const [key, value] of cache.entries()) {
    if ((now - value.timestamp) <= value.ttl) {
      cleanedCache.set(key, value);
    }
  }

  return cleanedCache;
}

/**
 * Limita el tamaño del caché eliminando las entradas más antiguas
 */
export function limitCacheSize<T>(
  cache: Map<string, T>,
  maxSize: number = CACHE_CONFIG.MAX_CACHE_SIZE
): Map<string, T> {
  if (cache.size <= maxSize) {
    return cache;
  }

  const entries = Array.from(cache.entries());
  entries.sort((a, b) => {
    // Asumir que T tiene una propiedad timestamp
    const aTime = (a[1] as any).timestamp || 0;
    const bTime = (b[1] as any).timestamp || 0;
    return aTime - bTime;
  });

  const toRemove = entries.length - maxSize;
  const limitedCache = new Map(cache);

  for (let i = 0; i < toRemove; i++) {
    limitedCache.delete(entries[i][0]);
  }

  return limitedCache;
}

/**
 * Debounce para evitar actualizaciones excesivas
 */
export function debounce<T extends (...args: any[]) => any>(
  func: T,
  wait: number
): (...args: Parameters<T>) => void {
  let timeout: NodeJS.Timeout;

  return (...args: Parameters<T>) => {
    clearTimeout(timeout);
    timeout = setTimeout(() => func(...args), wait);
  };
}

/**
 * Throttle para limitar la frecuencia de actualizaciones
 */
export function throttle<T extends (...args: any[]) => any>(
  func: T,
```

```
  limit: number
): (...args: Parameters<T>) => void {
  let inThrottle: boolean;

  return (...args: Parameters<T>) => {
    if (!inThrottle) {
      func(...args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  };
}

/**
 * Valida si un objeto es válido para el contexto
 */
export function isValidContextData(data: any): boolean {
  return data !== null && data !== undefined && typeof data === 'object';
}

/**
 * Clona profundamente un objeto para evitar mutaciones
 */
export function deepClone<T>(obj: T): T {
  if (obj === null || typeof obj !== 'object') {
    return obj;
  }

  if (obj instanceof Date) {
    return new Date(obj.getTime()) as T;
  }

  if (obj instanceof Array) {
    return obj.map(item => deepClone(item)) as T;
  }

  if (typeof obj === 'object') {
    const cloned = {} as T;
    for (const key in obj) {
      if (obj.hasOwnProperty(key)) {
        cloned[key] = deepClone(obj[key]);
      }
    }
    return cloned;
  }

  return obj;
}

/**
 * Crea un hash simple para un objeto
 */
export function createHash(obj: any): string {
  const str = JSON.stringify(obj);
  let hash = 0;

  for (let i = 0; i < str.length; i++) {
    const char = str.charCodeAt(i);
    hash = ((hash << 5) - hash) + char;
    hash = hash & hash; // Convertir a 32-bit integer
  }

  return hash.toString(36);
}

/**
 * Verifica si dos objetos son iguales (shallow comparison)
 */
export function shallowEqual(obj1: any, obj2: any): boolean {
  if (obj1 === obj2) return true;
```

```
218
219   if (obj1 == null || obj2 == null) return false;
220
221   if (typeof obj1 !== 'object' || typeof obj2 !== 'object') return false;
222
223   const keys1 = Object.keys(obj1);
224   const keys2 = Object.keys(obj2);
225
226   if (keys1.length !== keys2.length) return false;
227
228   for (let key of keys1) {
229     if (obj1[key] !== obj2[key]) return false;
230   }
231
232   return true;
233 }
234
```

Ø=ÜÁ shared\interfaces

Ø=ÜÄ shared\interfaces\plan-limitation-modal.interface.ts

```
1  export interface PlanLimitationModalData {
2    showModal: boolean;
3    modalType: 'upgrade' | 'blocked';
4    title: string;
5    message: string;
6    primaryButtonText: string;
7    secondaryButtonText?: string;
8    onPrimaryAction: () => void;
9    onSecondaryAction?: () => void;
10 }
11
```

Ø=ÜÁ shared\mobile-header

Ø=ÜÄ shared\mobile-header\mobile-header.component.ts

```
1  import { Component, OnDestroy } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { RouterModule, Router } from '@angular/router';
4  import { Store } from '@ngrx/store';
5  import { selectUser } from '../../features/auth/store/user.selectios';
6  import { setUserData } from '../../features/auth/store/user.actions';
7  import { UserStatus } from '../../features/overview/models/overview';
8  import { AuthService } from '../../features/auth/service/authService';
9
10 /**
11  * Mobile header component for responsive navigation.
12  *
13  * This component provides a mobile-friendly navigation header with a hamburger
14  * menu, user information, and logout functionality. It's designed for smaller
15  * screens where the sidebar is replaced by a collapsible header menu.
16  *
17  * Features:
18  * - Mobile menu toggle (hamburger menu)
```

```
 * - User information display (name, initials, admin status, ban status)
 * - Logout functionality
 * - Responsive design for mobile devices
 * - Menu open/close state management
 *
 * Relations:
 * - AuthService: Handles logout functionality
 * - Store (NgRx): Gets current user data
 * - Router: Navigation after logout
 *
 * @component
 * @selector app-mobile-header
 * @standalone true
 */
@Component({
  selector: 'app-mobile-header',
  standalone: true,
  imports: [CommonModule, RouterModule],
  templateUrl: './mobile-header.component.html',
  styleUrls: ['./mobile-header.component.scss']
})
export class MobileHeaderComponent implements OnDestroy {
  isMobileMenuOpen = false;
  userName: string = '';
  lastName: string = '';
  isAdmin: boolean = false;
  userToken: string = '';
  isBanned: boolean = false;

  constructor(
    private authService: AuthService,
    private router: Router,
    private store: Store
  ) {
    this.store.select(selectUser).subscribe((user) => {
      this.userName = user?.user?.firstName || '';
      this.lastName = user?.user?.lastName || '';
      this.isAdmin = user?.user?.isAdmin || false;
      this.userToken = user?.user?.tokenId || '';
      this.isBanned = user?.user?.status === UserStatus.BANNED;
    });
  }

  ngOnDestroy() {
    // Cleanup si es necesario
  }

  toggleMobileMenu() {
    this.isMobileMenuOpen = !this.isMobileMenuOpen;
  }

  closeMobileMenu() {
    this.isMobileMenuOpen = false;
  }

  onlyNameInitials(): string {
    if (!this.userName) return '';
    return this.userName.charAt(0) + this.lastName.charAt(1);
  }

  logout() {
    this.closeMobileMenu();
    this.authService
      .logout()
      .then(() => {
        // Limpiar todo el localStorage
        localStorage.clear();
        this.store.dispatch(setUserData({ user: null }));
        this.router.navigate(['/login']);
      })
```

```
      .catch((error) => {
        alert('Logout failed. Please try again.');
      });
  }
}

```

Ø=ÜÁ shared\pipes

Ø=ÜÄ shared\pipes\currency-format.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
import { NumberFormatterService } from '../utils/number-formatter.service';

/**
 * Pipe for formatting numbers as currency.
 *
 * This pipe transforms numeric values into formatted currency strings using
 * the NumberFormatterService. It handles null/undefined values gracefully
 * and formats values as USD currency with proper separators.
 *
 * Features:
 * - Formats numbers as currency (USD)
 * - Handles null/undefined values (returns '$0.00')
 * - Handles string inputs (converts to number)
 * - Uses NumberFormatterService for consistent formatting
 *
 * Usage:
 * {{ value | currencyFormat }}
 *
 * Relations:
 * - NumberFormatterService: Provides the actual formatting logic
 *
 * @pipe
 * @name currencyFormat
 * @standalone true
 */
@Pipe({
  name: 'currencyFormat',
  standalone: true
})
export class CurrencyFormatPipe implements PipeTransform {

  constructor(private numberFormatter: NumberFormatterService) {}

  transform(value: number | string | null | undefined): string {
    return this.numberFormatter.formatCurrency(value);
  }
}

```

Ø=ÜÄ shared\pipes\number-format.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
import { NumberFormatterService } from '../utils/number-formatter.service';

/**
 * Pipe for formatting numbers with thousand separators.
 *
```

```
 * This pipe transforms numeric values into formatted number strings with
 * proper thousand separators and decimal places. It uses the NumberFormatterService
 * for consistent formatting across the application.
 *
 * Features:
 * - Formats numbers with thousand separators
 * - Configurable decimal places (default: 2)
 * - Handles null/undefined values (returns '0.00')
 * - Handles string inputs (converts to number)
 *
 * Usage:
 * {{ value | numberFormat }}           // 2 decimals (default)
 * {{ value | numberFormat:0 }}         // 0 decimals
 * {{ value | numberFormat:4 }}         // 4 decimals
 *
 * Relations:
 * - NumberFormatterService: Provides the actual formatting logic
 *
 * @pipe
 * @name numberFormat
 * @standalone true
 */
@Pipe({
  name: 'numberFormat',
  standalone: true
})
export class NumberFormatPipe implements PipeTransform {

  constructor(private numberFormatter: NumberFormatterService) {}

  transform(value: number | string | null | undefined, decimals: number = 2): string {
    return this.numberFormatter.formatNumber(value, decimals);
  }
}

```

Ø=ÜÄ shared\pipes\percentage-format.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
import { NumberFormatterService } from '../utils/number-formatter.service';

/**
 * Pipe for formatting numbers as percentages.
 *
 * This pipe transforms numeric values into formatted percentage strings with
 * the % symbol. It uses the NumberFormatterService for consistent formatting
 * and handles the conversion from decimal to percentage format.
 *
 * Features:
 * - Formats numbers as percentages with % symbol
 * - Handles null/undefined values (returns '0.00%')
 * - Handles string inputs (converts to number)
 * - Converts decimal values to percentage (e.g., 0.5 !’ 50.00%)
 *
 * Usage:
 * {{ value | percentageFormat }}
 *
 * Relations:
 * - NumberFormatterService: Provides the actual formatting logic
 *
 * @pipe
 * @name percentageFormat
 * @standalone true
 */
@Pipe({
  name: 'percentageFormat',
```

```
  standalone: true
})
export class PercentageFormatPipe implements PipeTransform {

  constructor(private numberFormatter: NumberFormatterService) {}

  transform(value: number | string | null | undefined): string {
    return this.numberFormatter.formatPercentage(value);
  }
}

```

Ø=ÜÁ shared\pop-ups\alert-popup

Ø=ÜÄ shared\pop-ups\alert-popup\alert-popup.component.ts

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

/**
 * Component for displaying alert dialogs.
 *
 * This component provides a reusable alert dialog that can display different
 * types of messages (info, warning, error, success) with customizable title,
 * message, and button text.
 *
 * Features:
 * - Multiple alert types (info, warning, error, success)
 * - Customizable title and message
 * - Customizable button text
 * - Visibility control
 * - Close and confirm events
 *
 * Usage:
 * <app-alert-popup
 *   [visible]="showAlert"
 *   [title]="alertTitle"
 *   [message]="alertMessage"
 *   [type]="'error'"
 *   [buttonText]="'OK'"
 *   (close)="onCloseAlert()">
 * </app-alert-popup>
 *
 * Relations:
 * - AlertService: Often used together to show alerts
 *
 * @component
 * @selector app-alert-popup
 * @standalone true
 */
@Component({
  selector: 'app-alert-popup',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './alert-popup.component.html',
  styleUrl: './alert-popup.component.scss'
})
export class AlertPopupComponent {
  @Input() visible: boolean = false;
  @Input() title: string = 'Alert';
  @Input() message: string = '';
  @Input() buttonText: string = 'OK';
  @Input() type: 'info' | 'warning' | 'error' | 'success' = 'info';
```

```
48
49    @Output() close = new EventEmitter<void>();
50    @Output() confirm = new EventEmitter<void>();
51
52    onClose(): void {
53      this.close.emit();
54    }
55
56    onConfirm(): void {
57      this.confirm.emit();
58      this.close.emit();
59    }
60  }
61
```

Ø=ÜÁ shared\pop-ups\confirm-pop-up

Ø=ÜÄ shared\pop-ups\confirm-pop-up\confirm-popup.component.ts

```typescript
1  import { Component, Input } from '@angular/core';
2
3  import { CommonModule } from '@angular/common';
4
5  /**
6   * Component for displaying confirmation dialogs.
7   *
8   * This component provides a reusable confirmation dialog for actions that
9   * require user confirmation. It supports both regular and dangerous actions
10  * with customizable messages and button texts.
11  *
12  * Features:
13  * - Customizable title and message
14  * - Customizable confirm and cancel button texts
15  * - Dangerous action styling (for destructive actions)
16  * - Close and cancel callbacks
17  * - Visibility control
18  *
19  * Usage:
20  * <app-confirm-popup
21  *   [visible]="showConfirm"
22  *   [title]="'Confirm Action'"
23  *   [message]="'Are you sure?'"
24  *   [confirmButtonText]="'Confirm'"
25  *   [cancelButtonText]="'Cancel'"
26  *   [isDangerous]="false"
27  *   [close]="onClose"
28  *   [cancel]="onCancel">
29  * </app-confirm-popup>
30  *
31  * Relations:
32  * - Used by components that require user confirmation before actions
33  *
34  * @component
35  * @selector app-confirm-popup
36  * @standalone true
37  */
38 @Component({
39   selector: 'app-confirm-popup',
40   imports: [CommonModule],
41   templateUrl: './confirm-popup.component.html',
42   styleUrl: './confirm-popup.component.scss',
43   standalone: true,
44 })
```

```
export class ConfirmPopupComponent {
  @Input() visible = false;
  @Input() close!: () => void;
  @Input() cancel!: () => void;
  @Input() title: string = 'Confirm changes to your strategy?';
  @Input() message: string = 'These updates will modify how your trading rules are applied.';
  @Input() confirmButtonText: string = 'Apply changes';
  @Input() cancelButtonText: string = 'Cancel';
  @Input() isDangerous: boolean = false; // Para acciones destructivas (eliminar, etc.)
}

```

Ø=ÜÁ shared\pop-ups\edit-pop-up

Ø=ÜÄ shared\pop-ups\edit-pop-up\edit-popup.component.ts

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

/**
 * Component for displaying an edit popup overlay.
 *
 * This component provides a simple popup overlay that can be used to display
 * edit forms or content. It includes a close callback for handling dismissal.
 *
 * Features:
 * - Visibility control
 * - Close callback function
 * - Reusable popup overlay
 *
 * Usage:
 * <app-edit-popup
 *   [visible]="showEditPopup"
 *   [close]="onCloseEditPopup">
 * </app-edit-popup>
 *
 * Relations:
 * - Used by components that need edit popup overlays
 *
 * @component
 * @selector app-edit-popup
 * @standalone true
 */
@Component({
  selector: 'app-edit-popup',
  imports: [],
  templateUrl: './edit-popup.component.html',
  styleUrl: './edit-popup.component.scss',
  standalone: true,
})
export class EditPopupComponent {
  @Input() visible = false;
  @Input() close!: () => void;
}

```

Ø=ÜÁ shared\pop-ups\forgot-password

Ø=ÜÄ shared\pop-ups\forgot-password\forgot-password.component.ts

```typescript
import { Component, EventEmitter, Input, Output } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { AuthService } from '../../services/auth.service';

/**
 * Component for password reset functionality.
 *
 * This component provides a modal interface for users to request a password
 * reset email. It includes form validation, error handling, and success
 * messaging.
 *
 * Features:
 * - Email input with validation
 * - Password reset email sending
 * - Success and error message display
 * - Loading state management
 * - Form validation
 * - Close/back functionality
 *
 * Validation:
 * - Email is required
 * - Email must be valid format
 *
 * Error Handling:
 * - Handles user-not-found errors
 * - Handles invalid-email errors
 * - Generic error messages for other failures
 *
 * Relations:
 * - AuthService: Sends password reset email
 *
 * @component
 * @selector app-forgot-password-popup
 * @standalone true
 */
@Component({
  selector: 'app-forgot-password-popup',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './forgot-password.component.html',
  styleUrl: './forgot-password.component.scss'
})
export class ForgotPasswordPopupComponent {
  @Input() visible: boolean = false;
  @Output() close = new EventEmitter<void>();

  form: FormGroup;
  submitted = false;
  loading = false;
  successMessage = '';
  errorMessage = '';

  constructor(private fb: FormBuilder, private authService: AuthService) {
    this.form = this.fb.group({
      email: ['', [Validators.required, Validators.email]]
    });
  }

  onBack(): void {
    this.resetFeedback();
    this.close.emit();
  }

```

```
  async onSubmit(): Promise<void> {
    this.submitted = true;
    this.successMessage = '';
    this.errorMessage = '';
    if (this.form.invalid || this.loading) return;

    this.loading = true;
    const email = this.form.value.email as string;
    try {
      await this.authService.sendPasswordReset(email);
      this.successMessage = 'We have sent you an email with instructions.';
    } catch (error: any) {
      if (error?.code === 'auth/user-not-found') {
        this.errorMessage = 'No account found with this email.';
      } else if (error?.code === 'auth/invalid-email') {
        this.errorMessage = 'Invalid email format.';
      } else {
        this.errorMessage = 'Failed to send reset email. Please try again later.';
      }
    } finally {
      this.loading = false;
    }
  }

  private resetFeedback(): void {
    this.submitted = false;
    this.successMessage = '';
    this.errorMessage = '';
    this.loading = false;
  }
}
```

Ø=ÜÁ shared\pop-ups\loading-pop-up

Ø=ÜÄ shared\pop-ups\loading-pop-up\loading-popup.component.ts

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

/**
 * Component for displaying a full-screen loading overlay.
 *
 * This component provides a modal-style loading indicator that covers the entire
 * screen with a semi-transparent overlay. It displays a spinner and loading message
 * to indicate that an operation is in progress.
 *
 * Features:
 * - Full-screen overlay with semi-transparent background
 * - Animated spinner
 * - Loading message
 * - Visibility control via Input property
 * - Fixed positioning with high z-index
 *
 * Usage:
 * <app-loading-popup [visible]="isLoading"></app-loading-popup>
 *
 * Relations:
 * - Used by components that need to block user interaction during loading
 *
 * @component
```

```
 * @selector app-loading-popup
 * @standalone true
 */
@Component({
  selector: 'app-loading-popup',
  standalone: true,
  imports: [CommonModule],
  template: `
    <div class="loading-overlay" *ngIf="visible">
      <div class="loading-box">
        <div class="spinner"></div>
        <p>Loading data...</p>
      </div>
    </div>
  `,
  styles: [
    `
      .loading-overlay {
        position: fixed;
        top: 0;
        left: 0;
        right: 0;
        bottom: 0;
        background: rgba(0, 0, 0, 0.5);
        display: flex;
        align-items: center;
        justify-content: center;
        z-index: 2000;
      }
      .loading-box {
        background: #23252b;
        padding: 24px 36px;
        border-radius: 12px;
        color: #fff;
        display: flex;
        flex-direction: column;
        align-items: center;
      }
      .spinner {
        border: 4px solid rgba(255, 255, 255, 0.2);
        border-top: 4px solid #c8fc00;
        border-radius: 50%;
        width: 40px;
        height: 40px;
        animation: spin 1s linear infinite;
        margin-bottom: 12px;
      }
      @keyframes spin {
        0% {
          transform: rotate(0deg);
        }
        100% {
          transform: rotate(360deg);
        }
      }
    `,
  ],
})
export class LoadingPopupComponent {
  @Input() visible = false;
}

```

Ø=ÜÁ shared\pop-ups\stripe-loader-popup

Ø=ÜÄ shared\pop-ups\stripe-loader-popup\stripe-loader-popup.component.ts

```typescript
import { Component, Input } from '@angular/core';
import { CommonModule } from '@angular/common';

/**
 * Component for displaying a loading overlay during Stripe redirect.
 *
 * This component provides a full-screen loading overlay that is shown
 * when redirecting users to Stripe for payment processing. It displays
 * a loading message to inform users that they are being redirected.
 *
 * Features:
 * - Full-screen overlay
 * - Customizable loading message
 * - Visibility control
 *
 * Usage:
 * <app-stripe-loader-popup
 *   [visible]="isRedirecting"
 *   [message]="'Redirecting to Stripe...'">
 * </app-stripe-loader-popup>
 *
 * Relations:
 * - Used by PlanSettingsComponent during Stripe checkout redirect
 * - Used when opening Stripe customer portal
 *
 * @component
 * @selector app-stripe-loader-popup
 * @standalone true
 */
@Component({
  selector: 'app-stripe-loader-popup',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './stripe-loader-popup.component.html',
  styleUrl: './stripe-loader-popup.component.scss'
})
export class StripeLoaderPopupComponent {
  @Input() visible: boolean = false;
  @Input() message: string = 'Redirecting to Stripe...';
}

```

Ø=ÜÁ shared\services

Ø=ÜÄ shared\services\account-deletion.service.ts

```typescript
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';
import {
  getFirestore,
  collection,
  query,
  where,
  getDocs,
```

```
  deleteDoc,
  doc,
  writeBatch
} from 'firebase/firestore';
import { firebaseApp } from '../../firebase/firebase.init';

/**
 * Service for comprehensive user account deletion.
 *
 * This service handles the complete deletion of all user data from Firebase
 * when a user requests account deletion. It uses batch operations for atomic
 * deletion of all related data across multiple collections.
 *
 * Features:
 * - Delete all user data atomically (batch operations)
 * - Delete user accounts
 * - Delete user strategies (configuration-overview and configurations)
 * - Delete monthly reports
 * - Delete plugin history
 * - Delete link tokens
 * - Delete user subscriptions
 * - Delete user document
 *
 * Deletion Process:
 * 1. Collects all user data to delete
 * 2. Uses Firestore batch for atomic operations
 * 3. Deletes in order: accounts !’ strategies !’ reports !’ plugin history !’ tokens !’ subscription !’ user
 * 4. Returns success/failure status
 *
 * Data Deleted:
 * - `accounts`: All user trading accounts
 * - `configuration-overview`: Strategy metadata
 * - `configurations`: Strategy rules
 * - `monthly_reports`: Monthly trading reports
 * - `plugin_history`: Plugin activation history
 * - `tokens`: Link tokens
 * - `users/{userId}/subscription`: Subscription subcollection
 * - `users/{userId}`: User document
 *
 * Relations:
 * - Used by ProfileDetailsComponent for account deletion
 * - Ensures complete data removal for GDPR compliance
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class AccountDeletionService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Deletes all data associated with a user from Firebase
   * @param userId - ID of the user to delete
   * @returns Promise<boolean> - true if deleted successfully, false if there was an error
   */
  async deleteUserData(userId: string): Promise<boolean> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return false;
```

```
    }

    try {

      // Use batch for atomic operations
      const batch = writeBatch(this.db);
      let operationsCount = 0;

      // 1. Delete accounts
      const accountsDeleted = await this.deleteUserAccounts(userId, batch);
      operationsCount += accountsDeleted;

      // 2. Delete configuration-overview and associated configurations
      const configsDeleted = await this.deleteUserConfigurations(userId, batch);
      operationsCount += configsDeleted;

      // 3. Delete monthly_reports
      const reportsDeleted = await this.deleteUserMonthlyReports(userId, batch);
      operationsCount += reportsDeleted;

      // 4. Delete plugin_history
      const pluginHistoryDeleted = await this.deleteUserPluginHistory(userId, batch);
      operationsCount += pluginHistoryDeleted;

      // 5. Delete tokens
      const tokensDeleted = await this.deleteUserTokens(userId, batch);
      operationsCount += tokensDeleted;

      // 6. Delete user subscription subcollection
      const subscriptionsDeleted = await this.deleteUserSubscriptions(userId, batch);
      operationsCount += subscriptionsDeleted;

      // 7. Delete user from users collection
      const userDeleted = await this.deleteUser(userId, batch);
      operationsCount += userDeleted;

      // Execute all operations in batch
      if (operationsCount > 0) {
        await batch.commit();
        return true;
      } else {
        console.log(`& þ  No data found to delete for user ${userId}`);
        return true; // Not an error if no data exists
      }

    } catch (error) {
      console.error(''L Error deleting user data:', error);
      return false;
    }
  }

  /**
   * Deletes all user accounts
   */
  private async deleteUserAccounts(userId: string, batch: any): Promise<number> {
    try {
      const accountsRef = collection(this.db!, 'accounts');
      const q = query(accountsRef, where('userId', '==', userId));
      const querySnapshot = await getDocs(q);

      let count = 0;
      querySnapshot.forEach((docSnapshot) => {
        batch.delete(docSnapshot.ref);
        count++;
      });

      console.log(`Ø=ÜÊ Deleting ${count} accounts for user ${userId}`);
      return count;
    } catch (error) {
      console.error('Error deleting accounts:', error);
```

```
      return 0;
    }
  }

  /**
   * Deletes configuration-overview and associated configurations
   */
  private async deleteUserConfigurations(userId: string, batch: any): Promise<number> {
    try {
      const configOverviewRef = collection(this.db!, 'configuration-overview');
      const q = query(configOverviewRef, where('userId', '==', userId));
      const querySnapshot = await getDocs(q);

      let count = 0;
      const configIds: string[] = [];

      // First collect configurationId to delete configurations
      querySnapshot.forEach((docSnapshot) => {
        const data = docSnapshot.data();
        if (data['configurationId']) {
          configIds.push(data['configurationId']);
        }
        batch.delete(docSnapshot.ref);
        count++;
      });

      // Delete associated configurations
      for (const configId of configIds) {
        const configRef = doc(this.db!, 'configurations', configId);
        batch.delete(configRef);
        count++;
      }

      console.log(`&™þ  Deleting ${count} configurations for user ${userId}`);
      return count;
    } catch (error) {
      console.error('Error deleting configurations:', error);
      return 0;
    }
  }

  /**
   * Deletes user monthly_reports
   */
  private async deleteUserMonthlyReports(userId: string, batch: any): Promise<number> {
    try {
      const reportsRef = collection(this.db!, 'monthly_reports');
      const q = query(reportsRef, where('id', '==', userId));
      const querySnapshot = await getDocs(q);

      let count = 0;
      querySnapshot.forEach((docSnapshot) => {
        batch.delete(docSnapshot.ref);
        count++;
      });

      console.log(`Ø=ÜÈ Deleting ${count} monthly reports for user ${userId}`);
      return count;
    } catch (error) {
      console.error('Error deleting monthly reports:', error);
      return 0;
    }
  }

  /**
   * Deletes user plugin_history
   */
  private async deleteUserPluginHistory(userId: string, batch: any): Promise<number> {
    try {
      const pluginHistoryRef = collection(this.db!, 'plugin_history');
```

```
      const q = query(pluginHistoryRef, where('id', '==', userId));
      const querySnapshot = await getDocs(q);

      let count = 0;
      querySnapshot.forEach((docSnapshot) => {
        batch.delete(docSnapshot.ref);
        count++;
      });

      console.log(`Ø=Ý  Deleting ${count} plugin histories for user ${userId}`);
      return count;
    } catch (error) {
      console.error('Error deleting plugin history:', error);
      return 0;
    }
  }

  /**
   * Deletes user tokens
   */
  private async deleteUserTokens(userId: string, batch: any): Promise<number> {
    try {
      const tokensRef = collection(this.db!, 'tokens');
      const q = query(tokensRef, where('userId', '==', userId));
      const querySnapshot = await getDocs(q);

      let count = 0;
      querySnapshot.forEach((docSnapshot) => {
        batch.delete(docSnapshot.ref);
        count++;
      });

      console.log(`Ø=Ý  Deleting ${count} tokens for user ${userId}`);
      return count;
    } catch (error) {
      console.error('Error deleting tokens:', error);
      return 0;
    }
  }

  /**
   * Deletes user subscription subcollection
   */
  private async deleteUserSubscriptions(userId: string, batch: any): Promise<number> {
    try {
      const subscriptionsRef = collection(this.db!, 'users', userId, 'subscription');
      const querySnapshot = await getDocs(subscriptionsRef);

      let count = 0;
      querySnapshot.forEach((docSnapshot) => {
        batch.delete(docSnapshot.ref);
        count++;
      });

      console.log(`Ø=Ü³ Deleting ${count} subscriptions for user ${userId}`);
      return count;
    } catch (error) {
      console.error('Error deleting subscriptions:', error);
      return 0;
    }
  }

  /**
   * Deletes user from users collection
   */
  private async deleteUser(userId: string, batch: any): Promise<number> {
    try {
      const userRef = doc(this.db!, 'users', userId);
      batch.delete(userRef);
      console.log(`Ø=Üd Deleting user ${userId} from users collection`);
```

```
      return 1;
    } catch (error) {
      console.error('Error deleting user:', error);
      return 0;
    }
  }
}

```

Ø=ÜÄ shared\services\accounts-operations.service.ts

```
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { getFirestore, doc, setDoc, getDoc, collection, query, where, getDocs, deleteDoc, updateDoc } from 'firebase/firestore';
import { isPlatformBrowser } from '@angular/common';
import { AccountData } from '../../features/auth/models/userModel';

/**
 * Service for trading account operations in Firebase.
 *
 * This service provides CRUD operations for trading accounts, including
 * creation, retrieval, updates, and deletion. It also includes validation
 * methods to check for duplicate emails and account IDs.
 *
 * Features:
 * - Create trading account
 * - Get user accounts
 * - Get all accounts
 * - Check if email exists (for validation)
 * - Check if account ID exists (for validation)
 * - Update account
 * - Delete account (returns userId for cache invalidation)
 *
 * Account Validation:
 * - Checks for duplicate email addresses across users
 * - Checks for duplicate account IDs across users
 * - Excludes current user's accounts from duplicate checks
 *
 * Data Structure:
 * - Stored in: `accounts/{accountId}`
 * - Contains: Account credentials, broker info, balance, trading stats
 *
 * Relations:
 * - Used by AuthService for account management
 * - Used by TradingAccountsComponent for account operations
 * - Used by CreateAccountPopupComponent for account creation
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class AccountsOperationsService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
```

```
   * Crear cuenta de trading
   */
  async createAccount(account: AccountData): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }
    await setDoc(doc(this.db, 'accounts', account.id), account);
  }

  /**
   * Obtener cuentas de un usuario
   */
  async getUserAccounts(userId: string): Promise<AccountData[] | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }
    const accountsCollection = collection(this.db, 'accounts');
    const q = query(accountsCollection, where('userId', '==', userId));
    const querySnapshot = await getDocs(q);
    const accounts: AccountData[] = [];
    querySnapshot.forEach((doc) => {
      accounts.push(doc.data() as AccountData);
    });
    return accounts.length > 0 ? accounts : null;
  }

  /**
   * Obtener todas las cuentas
   */
  async getAllAccounts(): Promise<AccountData[] | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }
    const accountsCollection = collection(this.db, 'accounts');
    const querySnapshot = await getDocs(accountsCollection);
    const accounts: AccountData[] = [];
    querySnapshot.forEach((doc) => {
      accounts.push(doc.data() as AccountData);
    });
    return accounts.length > 0 ? accounts : null;
  }

  /**
   * Verificar si un email de trading ya existe
   */
  async checkEmailExists(emailTradingAccount: string, currentUserId: string):
Promise<boolean> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return false;
    }
    const accountsCollection = collection(this.db, 'accounts');
    const q = query(
      accountsCollection,
      where('emailTradingAccount', '==', emailTradingAccount),
      where('userId', '!=', currentUserId) // Exclude current user's accounts
    );
    const querySnapshot = await getDocs(q);
    return !querySnapshot.empty; // Returns true if email exists for another user
  }

  /**
   * Verificar si un accountID ya existe
   */
  async checkAccountIdExists(accountID: string, currentUserId: string): Promise<boolean> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return false;
```

```
    }
    const accountsCollection = collection(this.db, 'accounts');
    const q = query(
      accountsCollection,
      where('accountID', '==', accountID),
      where('userId', '!=', currentUserId) // Exclude current user's accounts
    );
    const querySnapshot = await getDocs(q);
    return !querySnapshot.empty; // Returns true if accountID exists for another user
  }

  /**
   * Actualizar cuenta
   */
  async updateAccount(accountId: string, accountData: AccountData): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }
    const accountDoc = doc(this.db, 'accounts', accountId);
    await updateDoc(accountDoc, {
      accountName: accountData.accountName,
      broker: accountData.broker,
      server: accountData.server,
      emailTradingAccount: accountData.emailTradingAccount,
      brokerPassword: accountData.brokerPassword,
      accountID: accountData.accountID,
      accountNumber: accountData.accountNumber,
      balance: accountData.balance,
      initialBalance: accountData.initialBalance,
      netPnl: accountData.netPnl,
      profit: accountData.profit,
      bestTrade: accountData.bestTrade,
    });
  }

  /**
   * Eliminar cuenta
   */
  async deleteAccount(accountId: string): Promise<string | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    // Obtener el userId antes de eliminar la cuenta
    const accountDoc = doc(this.db, 'accounts', accountId);
    const accountSnap = await getDoc(accountDoc);

    if (!accountSnap.exists()) {
      throw new Error('Account not found');
    }

    const accountData = accountSnap.data() as AccountData;
    const userId = accountData.userId || null;

    // Eliminar la cuenta
    await deleteDoc(accountDoc);

    // Retornar el userId para poder actualizar los conteos
    return userId;
  }

  /**
   * Verificar unicidad de cuenta (email y accountID)
   */
  async validateAccountUniqueness(emailTradingAccount: string, accountID: string,
currentUserId: string): Promise<{ isValid: boolean; message: string }> {
    try {
      const [emailExists, accountIdExists] = await Promise.all([
        this.checkEmailExists(emailTradingAccount, currentUserId),
```

```
        this.checkAccountIdExists(accountID, currentUserId)
      ]);

      if (emailExists || accountIdExists) {
        return {
          isValid: false,
          message: 'This account is already registered, try with another account or delete this trade account first'
        };
      }

      return {
        isValid: true,
        message: 'Account creation/update successful'
      };
    } catch (error) {
      console.error('Error validating account uniqueness:', error);
      return {
        isValid: false,
        message: 'Error validating account uniqueness'
      };
    }
  }

  /**
   * Obtener el número total de cuentas de trading de un usuario
   */
  async getAllLengthUserAccounts(userId: string): Promise<number> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return 0;
    }

    try {
      const accountsCollection = collection(this.db, 'accounts');
      const q = query(accountsCollection, where('userId', '==', userId));
      const querySnapshot = await getDocs(q);
      return querySnapshot.size;
    } catch (error) {
      console.error('Error getting accounts count:', error);
      return 0;
    }
  }

  /**
   * Verificar si existe una cuenta con la combinación broker + server + accountID
   */
  async checkAccountExists(broker: string, server: string, accountID: string, currentUserId: string): Promise<boolean> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return false;
    }

    try {
      const accountsRef = collection(this.db, 'accounts');
      const q = query(
        accountsRef,
        where('broker', '==', broker),
        where('server', '==', server),
        where('accountID', '==', accountID),
        where('userId', '!=', currentUserId) // Excluir la cuenta actual si estamos editando
      );

      const querySnapshot = await getDocs(q);
      return !querySnapshot.empty;
    } catch (error) {
      console.error('Error checking account existence:', error);
      return false;
    }
  }
}
```

Ø=ÜÄ shared\services\alert.service.ts

```typescript
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

/**
 * Configuration interface for alert dialogs.
 *
 * @interface AlertConfig
 */
export interface AlertConfig {
  title: string;
  message: string;
  buttonText?: string;
  type?: 'info' | 'warning' | 'error' | 'success';
}

/**
 * Service for displaying alert dialogs throughout the application.
 *
 * This service provides a centralized way to show alert messages with different
 * types (info, warning, error, success). It uses RxJS BehaviorSubject to manage
 * alert state and provides convenience methods for common alert types.
 *
 * Features:
 * - Show alerts with custom title, message, and type
 * - Convenience methods for error, warning, success, and info alerts
 * - Observable stream for alert state changes
 * - Hide alerts programmatically
 *
 * Usage:
 * Components can subscribe to `alert$` observable to display alerts, or use
 * the convenience methods that automatically show alerts.
 *
 * Relations:
 * - Used by components throughout the application for user notifications
 * - AlertPopupComponent: Displays the actual alert UI
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class AlertService {
  private alertSubject = new BehaviorSubject<{ visible: boolean; config: AlertConfig | null }
>({ visible: false,
    config: null
  });

  public alert$ = this.alertSubject.asObservable();

  showAlert(config: AlertConfig): void {
    this.alertSubject.next({
      visible: true,
      config: {
        title: config.title,
        message: config.message,
        buttonText: config.buttonText || 'OK',
        type: config.type || 'info'
      }
    });
  }
```

```
63
64    hideAlert(): void {
65      this.alertSubject.next({
66        visible: false,
67        config: null
68      });
69    }
70
71    // Métodos de conveniencia
72    showError(message: string, title: string = 'Error'): void {
73      this.showAlert({
74        title,
75        message,
76        type: 'error'
77      });
78    }
79
80    showWarning(message: string, title: string = 'Warning'): void {
81      this.showAlert({
82        title,
83        message,
84        type: 'warning'
85      });
86    }
87
88    showSuccess(message: string, title: string = 'Success'): void {
89      this.showAlert({
90        title,
91        message,
92        type: 'success'
93      });
94    }
95
96    showInfo(message: string, title: string = 'Information'): void {
97      this.showAlert({
98        title,
99        message,
100       type: 'info'
101     });
102   }
103 }
104
```

Ø=ÜÄ shared\services\auth.service.ts

```
1  import { isPlatformBrowser } from '@angular/common';
2  import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
3  import {
4    createUserWithEmailAndPassword,
5    getAuth,
6    GoogleAuthProvider,
7    OAuthProvider,
8    onAuthStateChanged,
9    signInWithEmailAndPassword,
10   signInWithPopup,
11 } from 'firebase/auth';
12 import { BehaviorSubject, filter, Observable } from 'rxjs';
13 import { AppContextService } from '../context';
14 import { PlanService, Plan } from './planService';
15 import { SubscriptionService, Subscription } from './subscription-service';
16 import { UserStatus } from '../../features/overview/models/overview';
17 import { UsersOperationsService } from './users-operations.service';
18 import { AccountsOperationsService } from './accounts-operations.service';
19 import { StrategyOperationsService } from './strategy-operations.service';
20 import { TokensOperationsService, LinkToken } from './tokens-operations.service';
21 import { User } from '../../features/overview/models/overview';
```

```
import { AccountData, UserCredentials } from '../../features/auth/models/userModel';

/**
 * Authentication service for Firebase Auth and user management.
 *
 * This service provides comprehensive authentication functionality including
 * user registration, login (email/password, Google, Apple), logout, password
 * reset, and user data management. It also manages user plan subscriptions
 * and integrates with AppContextService for global state management.
 *
 * Features:
 * - User registration and login (email/password, Google, Apple)
 * - Logout functionality
 * - Password reset
 * - Authentication state observables
 * - User data CRUD operations
 * - Account management (trading accounts)
 * - Strategy management
 * - Token management (link tokens)
 * - User plan subscription management
 * - Global plans loading
 * - Real-time subscription listener
 *
 * Plan Management:
 * - Listens to user subscription changes
 * - Updates AppContextService with plan data
 * - Handles banned, cancelled, and active subscription states
 * - Loads global plans on authentication
 *
 * Relations:
 * - UsersOperationsService: User data operations
 * - AccountsOperationsService: Trading account operations
 * - StrategyOperationsService: Strategy operations
 * - TokensOperationsService: Link token operations
 * - SubscriptionService: Subscription management
 * - PlanService: Plan information
 * - AppContextService: Global state management
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({ providedIn: 'root' })
export class AuthService {
  private isBrowser: boolean;
  private authStateSubject = new BehaviorSubject<boolean | null>(null);
  authStateChanged = this.authStateSubject.asObservable();

  constructor(
    @Inject(PLATFORM_ID) private platformId: Object,
    private usersOperationsService: UsersOperationsService,
    private accountsOperationsService: AccountsOperationsService,
    private strategyOperationsService: StrategyOperationsService,
    private tokensOperationsService: TokensOperationsService,
    private appContext: AppContextService,
    private planService: PlanService,
    private subscriptionService: SubscriptionService
  ) {
    this.isBrowser = isPlatformBrowser(this.platformId);

    if (this.isBrowser) {
      onAuthStateChanged(getAuth(), async (user) => {
        this.authStateSubject.next(user !== null);
        if (user?.uid) {
          await this.startUserPlanListener(user.uid);
        } else {
          this.stopUserPlanListener();
          this.appContext.setUserPlan(null);
        }
      });
```

```
    } else {
      this.authStateSubject.next(false);
    }
  }

  private subscriptionUnsubscribe: (() => void) | null = null;

  private async startUserPlanListener(userId: string): Promise<void> {
    this.stopUserPlanListener();

    // Cargar planes globales si no están cargados
    await this.loadGlobalPlansIfNeeded();

    this.subscriptionUnsubscribe = this.subscriptionService.listenToUserLatestSubscription(
      userId,
      async (subscription) => {
        await this.updateUserPlanFromSubscription(subscription);
      }
    );
  }

  private stopUserPlanListener(): void {
    if (this.subscriptionUnsubscribe) {
      this.subscriptionUnsubscribe();
      this.subscriptionUnsubscribe = null;
    }
  }

  private async loadGlobalPlansIfNeeded(): Promise<void> {
    // Verificar si los planes globales ya están cargados
    const currentPlans = this.appContext.globalPlans();
    if (currentPlans.length > 0) {
      return;
    }

    try {
      const plans = await this.planService.getAllPlans();
      this.appContext.setGlobalPlans(plans);
    } catch (error) {
      console.error(''L Error cargando planes globales:', error);
    }
  }

  private async updateUserPlanFromSubscription(subscription: Subscription | null):
Promise<void> {
    try {
      if (!subscription) {
        this.appContext.setUserPlan(null);
        return;
      }

      const status = subscription.status;

      // Estado baneado: bloquear uso
      if (status === UserStatus.BANNED) {
        this.appContext.setUserPlan({
          planId: subscription.planId,
          planName: 'Banned',
          maxAccounts: 0,
          maxStrategies: 0,
          features: [],
          isActive: false,
          expiresAt: subscription.periodEnd ? (subscription.periodEnd as any).toMillis?.
() ?? undefined : undefined,
          // Extensiones opcionales
          status: UserStatus.BANNED,
          price: '0'
        } as any);
        return;
      }

      // Estado cancelado: plan Free
```

```
      if (status === UserStatus.CANCELLED) {
        this.appContext.setUserPlan({
          planId: 'free',
          planName: 'Free',
          maxAccounts: 1,
          maxStrategies: 1,
          features: [],
          isActive: true,
          status: UserStatus.CANCELLED,
          price: '0'
        } as any);
        return;
      }

      // Activo: cargar plan y mapear límites
      const plan: Plan | undefined = await this.planService.getPlanById(subscription.planId);
      if (!plan) {
        // Si el plan no existe, tratar como sin plan
        this.appContext.setUserPlan(null);
        return;
      }

      this.appContext.setUserPlan({
        planId: plan.id,
        planName: plan.name,
        maxAccounts: plan.tradingAccounts ?? 1,
        maxStrategies: plan.strategies ?? 1,
        features: [],
        isActive: true,
        status,
        price: plan.price
      } as any);
    } catch (error) {
      console.error('Error actualizando user plan desde subscription:', error);
      this.appContext.setUserPlan(null);
    }
  }

  // Firebase Auth primitives
  getAuth() { return getAuth(); }
  register(user: UserCredentials) { return createUserWithEmailAndPassword(getAuth(), user.email, user.password); }
  login(user: UserCredentials) { return signInWithEmailAndPassword(getAuth(), user.email, user.password); }
  loginWithGoogle() { const provider = new GoogleAuthProvider(); return signInWithPopup(getAuth(), provider); }
  loginWithApple() { const provider = new OAuthProvider('apple.com'); return signInWithPopup(getAuth(), provider); }
  logout() { return getAuth().signOut(); }

  async sendPasswordReset(email: string): Promise<void> {
    const { sendPasswordResetEmail } = await import('firebase/auth');
    return sendPasswordResetEmail(getAuth(), email);
  }

  // Observabilidad de sesión
  isAuthenticated(): Observable<boolean> {
    return this.authStateSubject.asObservable().pipe(filter((state): state is boolean => state !== null));
  }

  getCurrentUser(): any {
    try {
      if (!this.isBrowser) return null;
      return getAuth().currentUser;
    } catch (error) {
      console.error('Error obteniendo usuario actual:', error);
      return null;
    }
  }

  async getBearerTokenFirebase(userId: string): Promise<string> {
    const token = await getAuth().currentUser?.getIdToken();
    if (!token) throw new Error('Token not found');
    return token;
```

```
  }

  // Users collection operations
  async getUserData(uid: string): Promise<User> {
    this.appContext.setLoading('user', true);
    this.appContext.setError('user', null);
    try {
      const userData = await this.usersOperationsService.getUserData(uid);

      // Actualizar conteos de trading_accounts y strategies
      await this.updateUserCounts(uid);

      // Obtener nuevamente los datos actualizados después de actualizar los conteos
      const updatedUserData = await this.usersOperationsService.getUserData(uid);

      this.appContext.setCurrentUser(updatedUserData);
      this.appContext.setLoading('user', false);
      return updatedUserData;
    } catch (error) {
      this.appContext.setLoading('user', false);
      this.appContext.setError('user', 'Error al obtener datos del usuario');
      throw error;
    }
  }

  async getUserById(userId: string): Promise<User | null> {
    return this.usersOperationsService.getUserById(userId);
  }

  async updateUser(userId: string, userData: Partial<User>): Promise<void> {
    return this.usersOperationsService.updateUser(userId, userData);
  }

  async createUser(user: User) { return this.usersOperationsService.createUser(user); }
  async createLinkToken(token: LinkToken) { return
this.tokensOperationsService.createLinkToken(token); }
  async deleteLinkToken(tokenId: string) { return
this.tokensOperationsService.deleteLinkToken(tokenId); }
  async deleteUser(userId: string) { return this.usersOperationsService.deleteUser(userId); }

  // Accounts operations
  async createAccount(account: AccountData) {
    this.appContext.setLoading('accounts', true);
    this.appContext.setError('accounts', null);
    try {
      await this.accountsOperationsService.createAccount(account);
      this.appContext.addAccount(account);

      // Actualizar conteos del usuario
      if (account.userId) {
        await this.updateUserCounts(account.userId);
      }

      this.appContext.setLoading('accounts', false);
    } catch (error) {
      this.appContext.setLoading('accounts', false);
      this.appContext.setError('accounts', 'Error al crear cuenta');
      throw error;
    }
  }

  async getUserAccounts(userId: string): Promise<AccountData[] | null> {
    this.appContext.setLoading('accounts', true);
    this.appContext.setError('accounts', null);
    try {
      const accounts = await this.accountsOperationsService.getUserAccounts(userId);
      this.appContext.setUserAccounts(accounts || []);
      this.appContext.setLoading('accounts', false);
      return accounts;
    } catch (error) {
      this.appContext.setLoading('accounts', false);
      this.appContext.setError('accounts', 'Error al obtener cuentas del usuario');
```

```
      throw error;
    }
  }

  async getAllAccounts(): Promise<AccountData[] | null> { return this.accountsOperationsService.getAllAccounts(); }
  async checkEmailExists(email: string, currentUserId: string): Promise<boolean> { return this.accountsOperationsService.checkEmailExists(email, currentUserId); }
  async checkAccountIDExists(accountID: string, currentUserId: string): Promise<boolean> { return this.accountsOperationsService.checkAccountIDExists(accountID, currentUserId); }
  async checkAccountExists(broker: string, server: string, accountId: string, currentUserId: string): Promise<boolean> { return this.accountsOperationsService.checkAccountExists(broker, server, accountId, currentUserId); }
  async updateAccount(accountId: string, accountData: AccountData): Promise<void> { return this.accountsOperationsService.updateAccount(accountId, accountData); }
  async deleteAccount(accountId:string): Promise<void> {
    const userId = await this.accountsOperationsService.deleteAccount(accountId);

    // Actualizar conteos del usuario después de eliminar la cuenta
    if (userId) {
      await this.updateUserCounts(userId);
    }
  }

  // Verificar si un email de usuario ya está registrado
  async getUserByEmail(email: string): Promise<User | null> {
    try {
      return await this.usersOperationsService.getUserByEmail(email);
    } catch (error) {
      console.error('Error checking if email exists:', error);
      return null;
    }
  }

  // Método para obtener datos del usuario para validaciones (cuentas y estrategias)
  async getUserDataForValidation(userId: string): Promise<{
    accounts: AccountData[];
    strategies: any[];
  }> {
    try {
      const [accounts, strategies] = await Promise.all([
        this.accountsOperationsService.getUserAccounts(userId),
        this.strategyOperationsService.getUserStrategyViews(userId)
      ]);

      return {
        accounts: accounts || [],
        strategies: strategies || []
      };
    } catch (error) {
      console.error('Error getting user data for validation:', error);
      return {
        accounts: [],
        strategies: []
      };
    }
  }

  /**
   * Actualizar los conteos de trading_accounts y strategies del usuario
   */
  async updateUserCounts(userId: string): Promise<void> {
    try {
      const [tradingAccountsCount, strategiesCount] = await Promise.all([
        this.accountsOperationsService.getAllLengthUserAccounts(userId),
        this.strategyOperationsService.getAllLengthConfigurationsOverview(userId)
      ]);

      await this.updateUser(userId, {
        trading_accounts: tradingAccountsCount,
        strategies: strategiesCount
      });
    } catch (error) {
      console.error('Error updating user counts:', error);
    }
  }
```

```
}
```

Ø=ÜÄ shared\services\countryService.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, map, shareReplay, tap } from 'rxjs';

/**
 * Interface for country data from REST Countries API.
 *
 * @interface Country
 */
export interface Country {
  name: {
    common: string;
    official: string;
  };
  flags: {
    png: string;
    svg: string;
    alt: string;
  };
  idd: {
    root: string;
    suffixes: string[];
  };
}

/**
 * Interface for formatted country option.
 *
 * @interface CountryOption
 */
export interface CountryOption {
  code: string;
  name: string;
  flag: string;
  dialCode: string;
}

/**
 * Service for fetching and managing country data.
 *
 * This service fetches country data from the REST Countries API and formats
 * it for use in phone number inputs. It includes country codes, flags, and
 * dial codes, with caching to avoid repeated API calls.
 *
 * Features:
 * - Fetch all countries from REST Countries API
 * - Format countries with codes, names, flags, and dial codes
 * - Cache countries data in memory
 * - Share replay for multiple subscribers
 * - Filter countries with valid dial codes
 * - Sort countries alphabetically
 * - Map special country names to ISO codes
 *
 * API:
 * - Source: https://restcountries.com/v3.1/all
 * - Fields: idd (dial codes), flags, name
 *
 * Caching:
```

```
 * - Caches countries after first fetch
 * - Uses shareReplay for multiple subscribers
 * - Prevents duplicate API calls
 *
 * Relations:
 * - Used by PhoneInputComponent for country selection
 * - Used by UserModalComponent for country detection
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class CountryService {
  private apiUrl = 'https://restcountries.com/v3.1/all?fields=idd,flags,name';
  private cachedCountries: CountryOption[] | null = null;
  private inFlight$?: Observable<CountryOption[]>;

  constructor(private http: HttpClient) {}

  getCountries(): Observable<CountryOption[]> {
    if (this.cachedCountries) {
      return new Observable<CountryOption[]>((subscriber) => {
        subscriber.next(this.cachedCountries as CountryOption[]);
        subscriber.complete();
      });
    }

    if (this.inFlight$) {
      return this.inFlight$;
    }

    this.inFlight$ = this.http.get<Country[]>(this.apiUrl).pipe(
      map(countries =>
        countries
          .filter(country => country.idd && country.idd.root && country.idd.suffixes)
          .map(country => ({
            code: this.extractCountryCode(country.name.common),
            name: country.name.common,
            flag: country.flags.svg,
            dialCode: this.formatDialCode(country.idd.root, country.idd.suffixes[0])
          }))
          .sort((a, b) => a.name.localeCompare(b.name))
      ),
      tap(list => {
        this.cachedCountries = list;
      }),
      shareReplay(1)
    );

    return this.inFlight$;
  }

  private extractCountryCode(countryName: string): string {
    // Mapeo de nombres de países a códigos ISO para casos especiales
    const countryCodeMap: { [key: string]: string } = {
      'United States': 'US',
      'United Kingdom': 'GB',
      'South Korea': 'KR',
      'North Korea': 'KP',
      'United Arab Emirates': 'AE',
      'Czech Republic': 'CZ',
      'Dominican Republic': 'DO',
      'Central African Republic': 'CF',
      'Republic of the Congo': 'CG',
      'Democratic Republic of the Congo': 'CD',
      'South Africa': 'ZA',
      'New Zealand': 'NZ',
```

```
        'Papua New Guinea': 'PG',
        'Saudi Arabia': 'SA',
        'Costa Rica': 'CR',
        'El Salvador': 'SV',
        'Puerto Rico': 'PR',
        'Trinidad and Tobago': 'TT',
        'Saint Vincent and the Grenadines': 'VC',
        'Saint Kitts and Nevis': 'KN',
        'Antigua and Barbuda': 'AG',
        'Saint Lucia': 'LC',
        'Dominica': 'DM',
        'Grenada': 'GD',
        'Barbados': 'BB',
        'Bahamas': 'BS',
        'Bermuda': 'BM',
        'Cayman Islands': 'KY',
        'British Virgin Islands': 'VG',
        'US Virgin Islands': 'VI',
        'Turks and Caicos Islands': 'TC',
        'Anguilla': 'AI',
        'Montserrat': 'MS',
        'Sint Maarten': 'SX',
        'Aruba': 'AW',
        'Curaçao': 'CW',
        'Bonaire': 'BQ',
        'Saba': 'BQ',
        'Sint Eustatius': 'BQ',
        'Falkland Islands': 'FK',
        'South Georgia and the South Sandwich Islands': 'GS',
        'British Indian Ocean Territory': 'IO',
        'Pitcairn Islands': 'PN',
        'Saint Helena': 'SH',
        'Ascension Island': 'AC',
        'Tristan da Cunha': 'TA',
        'Norfolk Island': 'NF',
        'Christmas Island': 'CX',
        'Cocos Islands': 'CC',
        'Heard Island and McDonald Islands': 'HM',
        'Bouvet Island': 'BV',
        'Svalbard and Jan Mayen': 'SJ',
        'Åland Islands': 'AX',
        'Faroe Islands': 'FO',
        'Greenland': 'GL',
        'French Guiana': 'GF',
        'Guadeloupe': 'GP',
        'Martinique': 'MQ',
        'Mayotte': 'YT',
        'Réunion': 'RE',
        'Saint Barthélemy': 'BL',
        'Saint Martin': 'MF',
        'Saint Pierre and Miquelon': 'PM',
        'Wallis and Futuna': 'WF',
        'French Polynesia': 'PF',
        'New Caledonia': 'NC',
        'Cook Islands': 'CK',
        'Niue': 'NU',
        'Tokelau': 'TK',
        'American Samoa': 'AS',
        'Guam': 'GU',
        'Northern Mariana Islands': 'MP',
        'Marshall Islands': 'MH',
        'Micronesia': 'FM',
        'Palau': 'PW',
        'Solomon Islands': 'SB',
        'Vanuatu': 'VU',
        'Fiji': 'FJ',
        'Tonga': 'TO',
        'Kiribati': 'KI',
        'Nauru': 'NR',
        'Tuvalu': 'TV',
```

```
        'Samoa': 'WS',
        'Timor-Leste': 'TL',
        'Brunei': 'BN',
        'Maldives': 'MV',
        'Sri Lanka': 'LK',
        'Bangladesh': 'BD',
        'Bhutan': 'BT',
        'Nepal': 'NP',
        'Myanmar': 'MM',
        'Thailand': 'TH',
        'Cambodia': 'KH',
        'Vietnam': 'VN',
        'Malaysia': 'MY',
        'Indonesia': 'ID',
        'Philippines': 'PH',
        'Taiwan': 'TW',
        'Hong Kong': 'HK',
        'Macau': 'MO',
        'Mongolia': 'MN',
        'Kazakhstan': 'KZ',
        'Uzbekistan': 'UZ',
        'Turkmenistan': 'TM',
        'Tajikistan': 'TJ',
        'Kyrgyzstan': 'KG',
        'Afghanistan': 'AF',
        'Pakistan': 'PK',
        'India': 'IN',
        'China': 'CN',
        'Japan': 'JP',
        'Russia': 'RU',
        'Belarus': 'BY',
        'Ukraine': 'UA',
        'Moldova': 'MD',
        'Romania': 'RO',
        'Bulgaria': 'BG',
        'Greece': 'GR',
        'Turkey': 'TR',
        'Cyprus': 'CY',
        'Lebanon': 'LB',
        'Syria': 'SY',
        'Iraq': 'IQ',
        'Iran': 'IR',
        'Israel': 'IL',
        'Palestine': 'PS',
        'Jordan': 'JO',
        'Kuwait': 'KW',
        'Qatar': 'QA',
        'Bahrain': 'BH',
        'Oman': 'OM',
        'Yemen': 'YE',
        'Egypt': 'EG',
        'Libya': 'LY',
        'Tunisia': 'TN',
        'Algeria': 'DZ',
        'Morocco': 'MA',
        'Western Sahara': 'EH',
        'Mauritania': 'MR',
        'Mali': 'ML',
        'Burkina Faso': 'BF',
        'Niger': 'NE',
        'Chad': 'TD',
        'Sudan': 'SD',
        'South Sudan': 'SS',
        'Ethiopia': 'ET',
        'Eritrea': 'ER',
        'Djibouti': 'DJ',
        'Somalia': 'SO',
        'Kenya': 'KE',
        'Uganda': 'UG',
        'Tanzania': 'TZ',
```

```
        'Rwanda': 'RW',
        'Burundi': 'BI',
        'Cameroon': 'CM',
        'Nigeria': 'NG',
        'Benin': 'BJ',
        'Togo': 'TG',
        'Ghana': 'GH',
        'Ivory Coast': 'CI',
        'Liberia': 'LR',
        'Sierra Leone': 'SL',
        'Guinea': 'GN',
        'Guinea-Bissau': 'GW',
        'Senegal': 'SN',
        'Gambia': 'GM',
        'Cape Verde': 'CV',
        'São Tomé and Príncipe': 'ST',
        'Equatorial Guinea': 'GQ',
        'Gabon': 'GA',
        'Angola': 'AO',
        'Zambia': 'ZM',
        'Zimbabwe': 'ZW',
        'Botswana': 'BW',
        'Namibia': 'NA',
        'Lesotho': 'LS',
        'Eswatini': 'SZ',
        'Madagascar': 'MG',
        'Mauritius': 'MU',
        'Seychelles': 'SC',
        'Comoros': 'KM',
        'Malawi': 'MW',
        'Mozambique': 'MZ',
        'Iceland': 'IS',
        'Ireland': 'IE',
        'Norway': 'NO',
        'Sweden': 'SE',
        'Finland': 'FI',
        'Denmark': 'DK',
        'Estonia': 'EE',
        'Latvia': 'LV',
        'Lithuania': 'LT',
        'Poland': 'PL',
        'Germany': 'DE',
        'Netherlands': 'NL',
        'Belgium': 'BE',
        'Luxembourg': 'LU',
        'France': 'FR',
        'Monaco': 'MC',
        'Liechtenstein': 'LI',
        'Switzerland': 'CH',
        'Austria': 'AT',
        'Slovakia': 'SK',
        'Hungary': 'HU',
        'Slovenia': 'SI',
        'Croatia': 'HR',
        'Bosnia and Herzegovina': 'BA',
        'Serbia': 'RS',
        'Montenegro': 'ME',
        'North Macedonia': 'MK',
        'Albania': 'AL',
        'Kosovo': 'XK',
        'Italy': 'IT',
        'San Marino': 'SM',
        'Vatican City': 'VA',
        'Malta': 'MT',
        'Spain': 'ES',
        'Portugal': 'PT',
        'Andorra': 'AD',
        'Canada': 'CA',
        'Mexico': 'MX',
        'Guatemala': 'GT',
```

```
      'Belize': 'BZ',
      'Honduras': 'HN',
      'Panama': 'PA',
      'Cuba': 'CU',
      'Jamaica': 'JM',
      'Haiti': 'HT',
      'Colombia': 'CO',
      'Venezuela': 'VE',
      'Guyana': 'GY',
      'Suriname': 'SR',
      'Brazil': 'BR',
      'Ecuador': 'EC',
      'Peru': 'PE',
      'Bolivia': 'BO',
      'Paraguay': 'PY',
      'Uruguay': 'UY',
      'Argentina': 'AR',
      'Chile': 'CL'
    };

    return countryCodeMap[countryName] || countryName.substring(0, 2).toUpperCase();
  }

  private formatDialCode(root: string, suffix: string): string {
    return root + suffix;
  }
}
```

Ø=ÜÄ shared\services\firebase-data.service.ts

```
import { getFirestore, collection, getDocs, doc, getDoc, setDoc, updateDoc, deleteDoc,
addDoc, query, where, orderBy, limit } from 'firebase/firestore';
import { isPlatformBrowser } from '@angular/common';
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';

/**
 * Generic service for Firebase Firestore operations.
 *
 * This service provides generic CRUD operations for any Firestore collection,
 * making it a utility service for common database operations. It abstracts
 * away the Firebase API details and provides a simple interface.
 *
 * Features:
 * - Get all documents from a collection
 * - Get document by ID
 * - Create document (with or without custom ID)
 * - Update document
 * - Delete document
 * - Query documents by field
 * - Check if document exists
 *
 * Usage:
 * Useful for simple operations that don't require specialized services.
 * For complex operations, use specific services (e.g., StrategyOperationsService).
 *
 * Relations:
 * - Used as a utility service throughout the application
 * - Provides generic database access patterns
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class FirebaseDataService {
```

```
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Generic method to get all documents from a collection
   */
  async getCollection(collectionName: string): Promise<any[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const snapshot = await getDocs(collection(this.db, collectionName));
      const documents: any[] = [];

      snapshot.forEach((doc) => {
        const data = doc.data();
        (data as any).id = doc.id;
        documents.push(data);
      });

      return documents;
    } catch (error) {
      console.error(`Error getting collection ${collectionName}:`, error);
      return [];
    }
  }

  /**
   * Generic method to get a document by ID
   */
  async getDocument(collectionName: string, docId: string): Promise<any | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const docRef = doc(this.db, collectionName, docId);
      const docSnap = await getDoc(docRef);

      if (docSnap.exists()) {
        const data = docSnap.data();
        (data as any).id = docSnap.id;
        return data;
      }
      return null;
    } catch (error) {
      console.error(`Error getting document ${docId} from ${collectionName}:`, error);
      return null;
    }
  }

  /**
   * Generic method to create a document
   */
  async createDocument(collectionName: string, data: any, docId?: string): Promise<string> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      throw new Error('Firestore not available');
    }
```

```
    try {
      if (docId) {
        await setDoc(doc(this.db, collectionName, docId), data);
        return docId;
      } else {
        const docRef = await addDoc(collection(this.db, collectionName), data);
        return docRef.id;
      }
    } catch (error) {
      console.error(`Error creating document in ${collectionName}:`, error);
      throw error;
    }
  }

  /**
   * Generic method to update a document
   */
  async updateDocument(collectionName: string, docId: string, data: any): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      await updateDoc(doc(this.db, collectionName, docId), data);
    } catch (error) {
      console.error(`Error updating document ${docId} in ${collectionName}:`, error);
      throw error;
    }
  }

  /**
   * Generic method to delete a document
   */
  async deleteDocument(collectionName: string, docId: string): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      await deleteDoc(doc(this.db, collectionName, docId));
    } catch (error) {
      console.error(`Error deleting document ${docId} from ${collectionName}:`, error);
      throw error;
    }
  }

  /**
   * Generic method to query documents
   */
  async queryDocuments(collectionName: string, field: string, operator: any, value: any):
Promise<any[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const q = query(
        collection(this.db, collectionName),
        where(field, operator, value)
      );

      const snapshot = await getDocs(q);
      const documents: any[] = [];

      snapshot.forEach((doc) => {
        const data = doc.data();
        (data as any).id = doc.id;
```

```
        documents.push(data);
      });

      return documents;
    } catch (error) {
      console.error(`Error querying documents from ${collectionName}:`, error);
      return [];
    }
  }

  /**
   * Check if a document exists
   */
  async documentExists(collectionName: string, docId: string): Promise<boolean> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return false;
    }

    try {
      const docRef = doc(this.db, collectionName, docId);
      const docSnap = await getDoc(docRef);
      return docSnap.exists();
    } catch (error) {
      console.error(`Error checking if document ${docId} exists in ${collectionName}:`,
error);return false;
    }
  }
}

```

Ø=ÜÄ shared\services\global-strategy-updater.service.ts

```
import { Injectable, Inject, PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';
import { StrategyDaysUpdaterService } from './strategy-days-updater.service';

/**
 * Service for global strategy updates.
 *
 * This service provides a high-level interface for updating strategy active
 * days. It acts as a wrapper around StrategyDaysUpdaterService, providing
 * convenient methods for updating all strategies or a single strategy.
 *
 * Features:
 * - Update all strategies for a user
 * - Update single strategy by ID
 * - Error handling and logging
 *
 * Relations:
 * - StrategyDaysUpdaterService: Performs the actual updates
 * - Used by components that need to refresh strategy days active
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class GlobalStrategyUpdaterService {
  private isBrowser: boolean;
  private updateInterval?: any;

  constructor(
    @Inject(PLATFORM_ID) private platformId: Object,
```

```
    private strategyDaysUpdater: StrategyDaysUpdaterService
  ) {
    this.isBrowser = isPlatformBrowser(this.platformId);
  }

  /**
   * Actualiza los días activos de la estrategia activa del usuario
   * @param userId - ID del usuario
   */
  async updateAllStrategies(userId: string): Promise<void> {
    if (!this.isBrowser) {
      return;
    }

    try {
      await this.strategyDaysUpdater.updateActiveStrategyDaysActive(userId);
    } catch (error) {
      console.error('GlobalStrategyUpdaterService: Error al actualizar estrategia activa:',
error);
    }
  }

  /**
   * Actualiza una estrategia específica
   * @param strategyId - ID de la estrategia
   * @param userId - ID del usuario
   */
  async updateSingleStrategy(strategyId: string, userId: string): Promise<void> {
    try {
      await this.strategyDaysUpdater.updateStrategyDaysActive(strategyId, userId);
    } catch (error) {
      console.error('GlobalStrategyUpdaterService: Error al actualizar estrategia:', error);
    }
  }
}

```

Ø=ÜÄ shared\services\monthly-reports.service.ts

```
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { getFirestore, collection, doc, getDoc, setDoc, updateDoc, deleteDoc, query, where,
orderBy, limit } from 'firebase/firestore';
import { isPlatformBrowser } from '@angular/common';
import { MonthlyReport } from '../../features/report/models/report.model';
import { newDataId } from '../../features/report/utils/firebase-data-utils';

/**
 * Service for managing monthly trading reports in Firebase.
 *
 * This service provides CRUD operations for monthly trading reports that
 * aggregate trading statistics by month. Reports are used for historical
 * analysis and performance tracking.
 *
 * Features:
 * - Update monthly report
 * - Get monthly report by ID
 * - Get all monthly reports
 * - Get monthly reports by user ID
 * - Get monthly report by user, month, and year
 * - Delete monthly report
 *
 * Report Structure:
 * - Stored in: `monthly_reports/{reportId}`
 * - Report ID format: Generated using `newDataId()` utility
 * - Contains: Monthly aggregated trading statistics
 *
 * Relations:
 * - Used by ReportService for saving monthly summaries
```

```
 * - Used by ReportComponent for historical data
 * - Uses `newDataId` utility for unique ID generation
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class MonthlyReportsService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Update monthly report
   */
  async updateMonthlyReport(monthlyReport: MonthlyReport): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      const id = newDataId(
        monthlyReport.id,
        monthlyReport.month,
        monthlyReport.year
      );

      await setDoc(doc(this.db, 'monthly_reports', id), monthlyReport);
    } catch (error) {
      console.error('Error updating monthly report:', error);
      throw error;
    }
  }

  /**
   * Get monthly report by ID
   */
  async getMonthlyReport(reportId: string): Promise<MonthlyReport | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const docRef = doc(this.db, 'monthly_reports', reportId);
      const docSnap = await getDoc(docRef);

      if (docSnap.exists()) {
        return docSnap.data() as MonthlyReport;
      }
      return null;
    } catch (error) {
      console.error('Error getting monthly report:', error);
      return null;
    }
  }

  /**
   * Get all monthly reports
```

```
   */
  async getAllMonthlyReports(): Promise<MonthlyReport[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const snapshot = await getDoc(doc(this.db, 'monthly_reports'));
      const reports: MonthlyReport[] = [];

      snapshot.data()?.['forEach']((doc: any) => {
        const data = doc.data() as MonthlyReport;
        (data as any).id = doc.id;
        reports.push(data);
      });

      return reports;
    } catch (error) {
      console.error('Error getting all monthly reports:', error);
      return [];
    }
  }

  /**
   * Get monthly reports by user ID
   */
  async getMonthlyReportsByUserId(userId: string): Promise<MonthlyReport[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const q = query(
        collection(this.db, 'monthly_reports'),
        where('userId', '==', userId),
        orderBy('year', 'desc'),
        orderBy('month', 'desc')
      );

      const snapshot = await getDoc(doc(this.db, 'monthly_reports'));
      const reports: MonthlyReport[] = [];

      snapshot.data()?.['forEach']((doc: any) => {
        const data = doc.data() as MonthlyReport;
        (data as any).id = doc.id;
        reports.push(data);
      });

      return reports;
    } catch (error) {
      console.error('Error getting monthly reports by user ID:', error);
      return [];
    }
  }

  /**
   * Get monthly report by user, month and year
   */
  async getMonthlyReportByUserMonthYear(userId: string, month: number, year: number):
Promise<MonthlyReport | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const q = query(
        collection(this.db, 'monthly_reports'),
        where('userId', '==', userId),
```

```
        where('month', '==', month),
        where('year', '==', year),
        limit(1)
      );

      const snapshot = await getDoc(doc(this.db, 'monthly_reports'));

      if (snapshot.exists()) {
        const doc = snapshot.data()?.[0];
        const data = doc.data() as MonthlyReport;
        (data as any).id = doc.id;
        return data;
      }

      return null;
    } catch (error) {
      console.error('Error getting monthly report by user, month and year:', error);
      return null;
    }
  }

  /**
   * Delete monthly report
   */
  async deleteMonthlyReport(reportId: string): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      await deleteDoc(doc(this.db, 'monthly_reports', reportId));
    } catch (error) {
      console.error('Error deleting monthly report:', error);
      throw error;
    }
  }
}
```

Ø=ÜÄ shared\services\overview-data.service.ts

```
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { getFirestore, collection, getDocs, query, where, orderBy, limit, startAfter, doc,
getDoc } from 'firebase/firestore';
import { isPlatformBrowser } from '@angular/common';

/**
 * Service for fetching overview dashboard data.
 *
 * This service provides methods to fetch data for the admin overview dashboard,
 * including users, subscriptions, monthly reports, strategies, and accounts.
 * It supports pagination for large datasets.
 *
 * Features:
 * - Get overview subscription data
 * - Get users data (with pagination)
 * - Get user accounts (with pagination)
 * - Get monthly reports data
 * - Get configuration overview data
 * - Get accounts data
 *
 * Pagination:
 * - Supports cursor-based pagination
 * - Orders users by subscription_date (descending)
 * - Orders accounts by accountID (descending)
 *
```

```
 * Relations:
 * - Used by OverviewService for data aggregation
 * - Used by OverviewComponent for dashboard display
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class OverviewDataService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Get overview subscription data
   */
  async getOverviewSubscriptionData() {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const snapshot = await getDocs(collection(this.db, 'overview-subscriptions'));
      return snapshot;
    } catch (error) {
      console.error('Error getting overview subscription data:', error);
      return null;
    }
  }

  /**
   * Get users data for overview
   */
  async getUsersData() {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const snapshot = await getDocs(collection(this.db, 'users'));
      return snapshot;
    } catch (error) {
      console.error('Error getting users data:', error);
      return null;
    }
  }

  /**
   * Paginación de usuarios para la tabla de Overview
   * Ordena por subscription_date desc (fallback: lastUpdated desc)
   */
  async getUsersPage(pageSize: number, startAfterDocId?: string) {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return { docs: [], lastDocId: undefined };
    }

    try {
```

```
      const usersCol = collection(this.db, 'users');
      let qRef: any = query(usersCol, orderBy('subscription_date', 'desc'), limit(pageSize));
      if (startAfterDocId) {
        const cursor = await getDoc(doc(this.db, 'users', startAfterDocId));
        if (cursor.exists()) {
          qRef = query(usersCol, orderBy('subscription_date', 'desc'), startAfter(cursor),
limit(pageSize));
        }
      }
      const snapshot = await getDocs(qRef);
      const lastDoc = snapshot.docs[snapshot.docs.length - 1];
      return { docs: snapshot.docs, lastDocId: lastDoc?.id };
    } catch (error) {
      console.error('Error getting users page:', error);
      return { docs: [], lastDocId: undefined };
    }
  }

  /**
   * Paginación de cuentas por usuario para la tabla (si se requiere desplegar cuentas)
   */
  async getUserAccountsPage(userId: string, pageSize: number, startAfterAccountId?: string) {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return { docs: [], lastDocId: undefined };
    }

    try {
      const accountsCol = collection(this.db, 'accounts');
      let qRef: any = query(
        accountsCol,
        where('userId', '==', userId),
        orderBy('accountID', 'desc'),
        limit(pageSize)
      );
      if (startAfterAccountId) {
        const cursor = await getDoc(doc(this.db, 'accounts', startAfterAccountId));
        if (cursor.exists()) {
          qRef = query(
            accountsCol,
            where('userId', '==', userId),
            orderBy('accountID', 'desc'),
            startAfter(cursor),
            limit(pageSize)
          );
        }
      }
      const snapshot = await getDocs(qRef);
      const lastDoc = snapshot.docs[snapshot.docs.length - 1];
      return { docs: snapshot.docs, lastDocId: lastDoc?.id };
    } catch (error) {
      console.error('Error getting user accounts page:', error);
      return { docs: [], lastDocId: undefined };
    }
  }

  /**
   * Get monthly reports data
   */
  async getMonthlyReportsData() {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const snapshot = await getDocs(collection(this.db, 'monthly_reports'));
      return snapshot;
    } catch (error) {
      console.error('Error getting monthly reports data:', error);
      return null;
```

```
    }
  }

  /**
   * Get configuration overview data
   */
  async getConfigurationOverviewData() {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const snapshot = await getDocs(collection(this.db, 'configuration-overview'));
      return snapshot;
    } catch (error) {
      console.error('Error getting configuration overview data:', error);
      return null;
    }
  }

  /**
   * Get accounts data
   */
  async getAccountsData() {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const snapshot = await getDocs(collection(this.db, 'accounts'));
      return snapshot;
    } catch (error) {
      console.error('Error getting accounts data:', error);
      return null;
    }
  }
}

```

Ø=ÜÄ shared\services\planService.ts

```
import { Injectable } from '@angular/core';
import { getFirestore, collection, query, getDocs, updateDoc, doc, Timestamp, setDoc,
orderBy, getDoc, deleteDoc, where, DocumentSnapshot } from 'firebase/firestore';
import { db } from '../../firebase/firebase.init';

/**
 * Interface for subscription plan data.
 *
 * @interface Plan
 */
export interface Plan {
  id: string;
  name: string;
  price: string;
  strategies: number;
  tradingAccounts: number;
  createdAt?: any;
  updatedAt?: any;
  planPriceId?: string;
}

/**
 * Service for managing subscription plans in Firebase.
 *
```

```
 * This service provides CRUD operations for subscription plans, including
 * creating, reading, updating, and deleting plans. Plans define the features
 * and limits available to users (e.g., number of strategies, trading accounts).
 *
 * Features:
 * - Create new plans
 * - Get all plans
 * - Get plan by ID
 * - Update existing plans
 * - Delete plans
 * - Query plans by name
 *
 * Plan Structure:
 * - Stored in: `plan/{planId}`
 * - Contains: name, price, strategies limit, trading accounts limit
 * - Includes Stripe price ID for payment integration
 *
 * Relations:
 * - Used by AuthService for loading global plans
 * - Used by PlanSettingsComponent for displaying available plans
 * - Used by PlanLimitationsGuard for checking plan limits
 * - Used by AppContextService for caching global plans
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class PlanService {

  /**
   * CREAR: Crear un nuevo plan
   * @param plan Datos del plan a crear
   * @returns Promise con el ID del documento creado
   */
  async createPlan(plan: Plan): Promise<string> {
    try {
      const planData = {
        ...plan,
        createdAt: new Date(),
        updatedAt: new Date()
      };

      await setDoc(doc(db, 'plan', planData.id), planData);
      return planData.id;
    } catch (error) {
      console.error(''L Error al crear plan:', error);
      throw error;
    }
  }

  /**
   * LEER: Obtener todos los planes
   * @returns Promise con array de todos los planes
   */
  async getAllPlans(): Promise<Plan[]> {
    try {
      const plansRef = collection(db, 'plan');

      // Intentar sin orderBy primero para ver si ese es el problema
      const querySnapshot = await getDocs(plansRef);

      if (querySnapshot.empty) {
        console.log('& þ  No se encontraron planes en la colección "plan"');
        return [];
      }

      const plans = querySnapshot.docs.map((doc) => {
```

```
        const data = doc.data();
        return { id: doc.id, ...data };
      });

      return plans as Plan[];
    } catch (error) {
      console.error('L Error al obtener planes:', error);
      return [];
    }
  }

  /**
   * LEER: Obtener un plan por ID
   * @param id ID del plan
   * @returns Promise con el plan específico
   */
  async getPlanById(id: string): Promise<Plan | undefined> {
    return getDoc(doc(db, 'plan', id))
      .then((doc) => ({ id: doc.id, ...doc.data() } as Plan))
      .catch((error) => {
        console.error('L Error al obtener el plan por ID:', error);
        return undefined;
      })
  }

  /**
   * ACTUALIZAR: Actualizar un plan existente
   * @param id ID del plan a actualizar
   * @param plan Datos actualizados del plan
   * @returns Promise que se resuelve cuando se completa la actualización
   */
  async updatePlan(id: string, plan: Partial<Omit<Plan, 'id' | 'createdAt'>>): Promise<void>
{   try {
      const updateData = {
        ...plan,
        updatedAt: new Date()
      };

      await updateDoc(doc(db, 'plan', id), updateData);
    } catch (error) {
      console.error('L Error al actualizar plan:', error);
      throw error;
    }
  }

  /**
   * ELIMINAR: Eliminar un plan
   * @param id ID del plan a eliminar
   * @returns Promise que se resuelve cuando se completa la eliminación
   */
  async deletePlan(id: string): Promise<void> {
    try {
      await deleteDoc(doc(db, 'plan', id));
    } catch (error) {
      console.error('L Error al eliminar plan:', error);
      throw error;
    }
  }

  /**
   * LEER: Obtener un plan por nombre exacto
   * @param name Nombre exacto del plan
   * @returns Promise con el plan específico o undefined si no se encuentra
   */
  async getPlanByName(name: string): Promise<Plan | undefined> {
    try {
      const plansRef = collection(db, 'plan');
      const querySnapshot = await getDocs(plansRef);

      if (querySnapshot.empty) {
```

```
        return undefined;
      }

      // Buscar plan con nombre exacto
      const matchingDoc = querySnapshot.docs.find(doc => {
        const data = doc.data();
        return data['name'] === name;
      });

      if (matchingDoc) {
        const data = matchingDoc.data();
        return { id: matchingDoc.id, ...data } as Plan;
      }

      return undefined;
    } catch (error) {
      console.error(''L Error al obtener plan por nombre:', error);
      return undefined;
    }
  }

  /**
   * LEER: Buscar planes por nombre
   * @param name Nombre o parte del nombre a buscar
   * @returns Promise con planes que coinciden con la búsqueda
   */
  async searchPlansByName(name: string): Promise<Plan[]> {
    try {

      const plansRef = collection(db, 'plan');

      // Primero obtener todos los planes para debug
      const allPlansSnapshot = await getDocs(plansRef);

      if (allPlansSnapshot.empty) {
        console.log('& þ  La colección "plan" está vacía');
        return [];
      }

      // Mostrar todos los planes disponibles
      allPlansSnapshot.docs.forEach((doc, index) => {
        const data = doc.data();
      });

      // Buscar planes que coincidan con el nombre (búsqueda simple)
      const matchingPlans = allPlansSnapshot.docs.filter(doc => {
        const data = doc.data();
        const planName = data['name']?.toLowerCase() || '';
        const searchName = name.toLowerCase();
        return planName.includes(searchName) || searchName.includes(planName);
      });

      const plans = matchingPlans.map((doc) => {
        const data = doc.data();
        console.log('  Plan encontrado:', { id: doc.id, name: data['name'] });
        return { id: doc.id, ...data };
      });

      return plans as Plan[];

    } catch (error) {
      console.error(''L Error en searchPlansByName:', error);
      return [];
    }
  }

  /**
   * UTILIDAD: Verificar si un plan existe
   * @param id ID del plan a verificar
   * @returns Promise que se resuelve con true si existe, false si no
```

```
   */
  async planExists(id: string): Promise<boolean> {
    try {
      const document = await getDoc(doc(db, 'plan', id)) as DocumentSnapshot<Plan>;
      return document.exists() || false;
    } catch (error) {
      console.error(''L Error al verificar existencia del plan:', error);
      return false;
    }
  }

  /**
   * UTILIDAD: Obtener el conteo total de planes
   * @returns Promise con el número total de planes
   */
  async getPlansCount(): Promise<number> {
    return getDocs(query(collection(db, 'plan')))
      .then((snapshot) => snapshot.docs.length)
      .catch((error) => {
        return 0;
      })
  }
}
```

Ø=ÜÄ shared\services\plugin-history.service.ts

```
import { isPlatformBrowser } from '@angular/common';
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import {
    collection,
    getDocs,
    getFirestore,
    where,
    query,
    onSnapshot,
    doc,
    getDoc,
} from 'firebase/firestore';
import { Observable, from } from 'rxjs';
import { firebaseApp } from '../../firebase/firebase.init';
import { TimezoneService } from './timezone.service';

/**
 * Interface for plugin history data.
 *
 * @interface PluginHistory
 */
export interface PluginHistory {
    id: string;
    isActive: boolean;
    updatedOn: string;
    tokenNeeded: boolean;
    dateActive: string[];
    dateInactive: string[];
}

/**
 * Service for managing plugin activation history.
 *
 * This service tracks when the trading plugin was activated and deactivated
 * for users. It's used to determine if trades were executed while the plugin
 * was active, which is essential for strategy adherence calculations.
 *
 * Features:
 * - Get plugin usage history for a user
 * - Determine if plugin is currently active based on dates
```

```
 * - Real-time listener for plugin history changes
 * - Timezone-aware date comparisons
 *
 * Plugin Status Logic:
 * - If `dateActive` has more elements than `dateInactive`: plugin is active
 * - If same count: compare last dates (last active > last inactive = active)
 * - Uses UTC conversion for accurate date comparisons
 *
 * Data Structure:
 * - Stored in: `plugin_history/plugin_{userId}`
 * - Contains: activation/deactivation date arrays
 *
 * Relations:
 * - Used by CalendarComponent for strategy adherence checks
 * - Used by ReportComponent for determining if trades followed strategies
 * - TimezoneService: For accurate date comparisons
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
    providedIn: 'root'
})
export class PluginHistoryService {

    private isBrowser: boolean;
    private db: ReturnType<typeof getFirestore> | null = null;

    constructor(
        @Inject(PLATFORM_ID) private platformId: Object,
        private timezoneService: TimezoneService
    ) {
        this.isBrowser = isPlatformBrowser(this.platformId);
        if (this.isBrowser) {
            this.db = getFirestore(firebaseApp);
        }
    }

    async getPluginUsageHistory(userId: string): Promise<PluginHistory[]> {
        if (!this.db) {
            console.warn('Firestore not available in SSR');
            return [];
        }

        try {
            // NUEVA LÓGICA: Buscar por document ID = plugin_{userId}
            const pluginDocId = `plugin_${userId}`;
            const docRef = doc(this.db, 'plugin_history', pluginDocId);
            const docSnap = await getDoc(docRef);

            if (!docSnap.exists()) {
                return [];
            }

            const data = docSnap.data();
            const pluginHistory = { id: docSnap.id, ...data };

            return [pluginHistory] as PluginHistory[];

        } catch (error) {
            console.error('Error getting plugin usage history:', error);
            return [];
        }

    }

    /**
     * MÉTODO NUEVO: Determinar si el plugin está activo basándose en las fechas
     * LÓGICA MEJORADA CON ZONA HORARIA:
```

```
     * - Si dateActive tiene más elementos que dateInactive: está activo
     * - Si tienen la misma cantidad: comparar la última fecha de cada array
     * - Si la última fecha de dateActive > última fecha de dateInactive: está activo
     * - Si la última fecha de dateInactive > última fecha de dateActive: está inactivo
     * - Usa conversión UTC para comparaciones precisas
     */
    isPluginActiveByDates(pluginHistory: PluginHistory): boolean {
        if (!pluginHistory.dateActive || !pluginHistory.dateInactive) {
            // Fallback al campo isActive si no hay fechas
            return pluginHistory.isActive;
        }

        const activeDates = pluginHistory.dateActive;
        const inactiveDates = pluginHistory.dateInactive;

        // Si dateActive tiene más elementos que dateInactive, está activo
        if (activeDates.length > inactiveDates.length) {
            return true;
        }

        // Si tienen la misma cantidad, comparar las últimas fechas
        if (activeDates.length === inactiveDates.length) {
            if (activeDates.length === 0) {
                return false; // No hay fechas, asumir inactivo
            }

            // MEJORA: Usar conversión UTC para comparaciones precisas
            const lastActiveDate =
this.timezoneService.convertToUTC(activeDates[activeDates.length - 1]);
            const lastInactiveDate =
this.timezoneService.convertToUTC(inactiveDates[inactiveDates.length - 1]);

            // Si la última fecha de active es mayor que la de inactive, está activo
            return lastActiveDate > lastInactiveDate;
        }

        // Si dateInactive tiene más elementos que dateActive, está inactivo
        return false;
    }

    /**
     * MÉTODO NUEVO: Listener en tiempo real para plugin history
     * FLUJO DINÁMICO:
     * - Retorna un Observable que emite cambios en tiempo real
     * - Filtra por userId específico
     * - El componente se suscribe y recibe actualizaciones automáticas
     * - Maneja errores y limpieza de recursos
     */
    getPluginHistoryRealtime(userId: string): Observable<PluginHistory[]> {
        if (!this.db) {
            console.warn('Firestore not available in SSR');
            return from([]);
        }

        return new Observable<PluginHistory[]>(subscriber => {
            // NUEVA LÓGICA: Buscar por document ID = plugin_{userId}
            const pluginDocId = `plugin_${userId}`;
            const pluginHistoryRef = collection(this.db!, 'plugin_history');
            const q = query(pluginHistoryRef, where('__name__', '==', pluginDocId));

            const unsubscribe = onSnapshot(q,
                (snapshot) => {
                    const pluginHistory = snapshot.docs.map(doc => ({
                        id: doc.id,
                        ...doc.data()
                    } as PluginHistory));

                    subscriber.next(pluginHistory);
                },
                (error) => {
                    console.error('Error in plugin history listener:', error);
```

```
                    subscriber.error(error);
                }
            );

            // Retornar función de limpieza
            return () => {
                unsubscribe();
            };
        });
    }

}
```

Ø=ÜÄ shared\services\reasons.service.ts

```typescript
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';
import {
  getFirestore,
  collection,
  addDoc,
  updateDoc,
  getDocs,
  doc,
  query,
  where,
  orderBy,
  limit,
  serverTimestamp,
} from 'firebase/firestore';

/**
 * Interface for ban reason record data.
 *
 * @interface BanReasonRecord
 */
export interface BanReasonRecord {
  id?: string;
  reason: string;
  dateBan: any; // serverTimestamp
  dateUnban: any | null; // serverTimestamp or null
}

/**
 * Service for managing user ban reasons.
 *
 * This service handles the creation and tracking of ban reasons for users.
 * It stores ban records in a subcollection under each user's document,
 * allowing administrators to track why users were banned and when they
 * were unbanned.
 *
 * Features:
 * - Create ban reason record
 * - Update ban reason (e.g., add unban date)
 * - Get latest open ban reason
 *
 * Data Structure:
 * - Stored in: `users/{userId}/reasons/{reasonId}`
 * - Tracks: ban reason, ban date, unban date
 *
 * Relations:
 * - Used by UsersDetailsComponent for ban/unban operations
 * - Used by AuthGuard for checking ban status
 *
 * @service
 * @injectable
```

```
 * @providedIn root
 */
@Injectable({ providedIn: 'root' })
export class ReasonsService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  private reasonsCollectionPath(userId: string) {
    if (!this.db) throw new Error('Firestore not available in SSR');
    return collection(this.db, 'users', userId, 'reasons');
  }

  async createReason(userId: string, reason: string): Promise<string> {
    const colRef = this.reasonsCollectionPath(userId);
    const docRef = await addDoc(colRef, {
      reason,
      dateBan: serverTimestamp(),
      dateUnban: null,
    } as BanReasonRecord);
    return docRef.id;
  }

  async updateReason(userId: string, reasonId: string, data: Partial<BanReasonRecord>):
Promise<void> {
    if (!this.db) throw new Error('Firestore not available in SSR');
    await updateDoc(doc(this.db, 'users', userId, 'reasons', reasonId), data as any);
  }

  async getOpenLatestReason(userId: string): Promise<BanReasonRecord | null> {
    // Para evitar índices compuestos, ordenamos por dateBan y tomamos el más reciente
    const colRef = this.reasonsCollectionPath(userId);
    const qRef = query(colRef, orderBy('dateBan', 'desc'), limit(1));
    const snapshot = await getDocs(qRef);
    if (snapshot.empty) return null;
    const docSnap = snapshot.docs[0];
    const data = docSnap.data() as BanReasonRecord;
    return { ...data, id: docSnap.id };
  }
}
```

Ø=ÜÄ shared\services\strategy-days-updater.service.ts

```
import { Injectable, Inject, PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';
import { getFirestore, collection, query, getDocs, updateDoc, doc, Timestamp, where } from
'firebase/firestore';import { firebaseApp } from '../../firebase/firebase.init';

/**
 * Service for updating strategy active days in Firebase.
 *
 * This service calculates and updates the number of days a strategy has
 * been active based on its creation date. It supports updating all user
 * strategies or a specific strategy.
 *
 * Features:
 * - Update active days for all user strategies
```

```
 * - Update active days for active strategy only
 * - Update active days for specific strategy
 * - Calculate days active from creation date
 *
 * Days Active Calculation:
 * - Calculates days from `created_at` timestamp to current date
 * - Updates `days_active` field in configuration-overview
 * - Automatically updates `updated_at` timestamp
 *
 * Relations:
 * - Used by GlobalStrategyUpdaterService for batch updates
 * - Used by StrategyCardComponent for displaying days active
 * - Updates `configuration-overview` collection
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class StrategyDaysUpdaterService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Updates active days for all user strategies
   * @param userId - User ID
   */
  async updateAllStrategiesDaysActive(userId: string): Promise<void> {
    if (!this.isBrowser || !this.db) {
      console.warn('StrategyDaysUpdaterService: Cannot execute on server');
      return;
    }

    try {
      // Get all user strategies
      const strategiesRef = collection(this.db, 'configuration-overview');
      const q = query(strategiesRef);
      const querySnapshot = await getDocs(q);

      const strategiesToUpdate: { id: string; daysActive: number }[] = [];

      querySnapshot.forEach((docSnapshot) => {
        const data = docSnapshot.data();

        // Verify that the strategy belongs to the user
        if (data['userId'] === userId && data['created_at']) {
          const daysActive = this.calculateDaysActive(data['created_at']);

          // Always update to keep synchronized
          strategiesToUpdate.push({
            id: docSnapshot.id,
            daysActive: daysActive
          });
        }
      });

      // Update all strategies
      const updatePromises = strategiesToUpdate.map(strategy =>
        updateDoc(doc(this.db!, 'configuration-overview', strategy.id), {
          days_active: strategy.daysActive,
          updated_at: Timestamp.now()
```

```
        })
      );

      if (updatePromises.length > 0) {
        await Promise.all(updatePromises);
      }

    } catch (error) {
      console.error('StrategyDaysUpdaterService: Error updating active days:', error);
      throw error;
    }
  }

  /**
   * Updates active days for the user's active strategy
   * @param userId - User ID
   */
  async updateActiveStrategyDaysActive(userId: string): Promise<void> {
    if (!this.isBrowser || !this.db) {
      console.warn('StrategyDaysUpdaterService: Cannot execute on server');
      return;
    }

    try {
      // Find the user's active strategy
      const strategiesRef = collection(this.db, 'configuration-overview');
      const q = query(
        strategiesRef,
        where('userId', '==', userId),
        where('status', '==', true)
      );
      const querySnapshot = await getDocs(q);

      if (querySnapshot.empty) {
        console.log('StrategyDaysUpdaterService: No active strategy found for user:',
userId);return;
      }

      // There should only be one active strategy
      const activeStrategyDoc = querySnapshot.docs[0];
      const data = activeStrategyDoc.data();

      if (!data['created_at']) {
        console.warn('StrategyDaysUpdaterService: Active strategy without creation date');
        return;
      }

      const daysActive = this.calculateDaysActive(data['created_at']);

      // Only update if days have changed
      if (data['days_active'] !== daysActive) {
        await updateDoc(activeStrategyDoc.ref, {
          days_active: daysActive,
          updated_at: Timestamp.now()
        });
        console.log(`StrategyDaysUpdaterService: Updated active strategy
${activeStrategyDoc.id} with ${daysActive} active days`);
      }

    } catch (error) {
      console.error('StrategyDaysUpdaterService: Error updating active strategy days:',
error);throw error;
    }
  }

  /**
   * Updates active days for a specific strategy
   * @param strategyId - Strategy ID
   * @param userId - User ID (for security verification)
   */
  async updateStrategyDaysActive(strategyId: string, userId: string): Promise<void> {
```

```
    if (!this.isBrowser || !this.db) {
      console.warn('StrategyDaysUpdaterService: Cannot execute on server');
      return;
    }

    try {
      const strategyRef = doc(this.db, 'configuration-overview', strategyId);
      const strategyDoc = await getDocs(query(collection(this.db, 'configuration-
overview')));
      let strategyData: any = null;
      strategyDoc.forEach(docSnapshot => {
        if (docSnapshot.id === strategyId && docSnapshot.data()['userId'] === userId) {
          strategyData = docSnapshot.data();
        }
      });

      if (!strategyData || !strategyData['created_at']) {
        console.warn('StrategyDaysUpdaterService: Strategy not found or without creation
date'); return;
      }

      const daysActive = this.calculateDaysActive(strategyData['created_at']);

      // Only update if days have changed
      if (strategyData['days_active'] !== daysActive) {
        await updateDoc(strategyRef, {
          days_active: daysActive,
          updated_at: Timestamp.now()
        });
      }

    } catch (error) {
      console.error('StrategyDaysUpdaterService: Error updating strategy active days:',
error);throw error;
    }
  }

  /**
   * Calculates active days since creation date
   * @param createdAt - Firebase timestamp or creation date
   * @returns Number of active days
   */
  private calculateDaysActive(createdAt: any): number {
    let createdDate: Date;

    // Handle different Firebase timestamp types
    if (createdAt && typeof createdAt.toDate === 'function') {
      // It's a Firebase Timestamp
      createdDate = createdAt.toDate();
    } else if (createdAt && createdAt.seconds) {
      // It's an object with seconds
      createdDate = new Date(createdAt.seconds * 1000);
    } else if (createdAt instanceof Date) {
      // Already a date
      createdDate = createdAt;
    } else if (typeof createdAt === 'string') {
      // It's a date string
      createdDate = new Date(createdAt);
    } else {
      console.warn('StrategyDaysUpdaterService: Unrecognized date format:', createdAt);
      return 0;
    }

    // Get current date and creation date in YYYY-MM-DD format (without hours)
    const now = new Date();
    const today = new Date(now.getFullYear(), now.getMonth(), now.getDate());
    const createdDay = new Date(createdDate.getFullYear(), createdDate.getMonth(),
createdDate.getDate());
    // Calculate difference in complete days
    const diffTime = today.getTime() - createdDay.getTime();
```

```
    const diffDays = Math.floor(diffTime / (1000 * 60 * 60 * 24));

    // If it's the same day, return 0
    // If complete days have passed, return the difference
    return Math.max(0, diffDays);
  }

  /**
   * Gets active days of a strategy without updating in Firebase
   * @param createdAt - Firebase timestamp or creation date
   * @returns Number of active days
   */
  getDaysActive(createdAt: any): number {
    return this.calculateDaysActive(createdAt);
  }
}
```

Ø=ÜÄ shared\services\strategy-operations.service.ts

```
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { getFirestore, doc, setDoc, getDoc, collection, query, where, getDocs, deleteDoc,
updateDoc, orderBy, limit, addDoc } from 'firebase/firestore';
import { isPlatformBrowser } from '@angular/common';
import { Timestamp } from 'firebase/firestore';
import { MaxDailyTradesConfig, StrategyState, ConfigurationOverview } from '../../features/
strategy/models/strategy.model';
/**
 * Service for strategy operations in Firebase.
 *
 * This service provides comprehensive CRUD operations for trading strategies,
 * managing both strategy metadata (configuration-overview) and strategy rules
 * (configurations). It handles the complete strategy lifecycle.
 *
 * Features:
 * - Configuration Overview Operations:
 *   - Create, read, update, delete strategy metadata
 *   - Get all strategies for a user
 *   - Get active strategies
 *   - Soft delete strategies (mark as deleted)
 * - Configuration Operations:
 *   - Create, read, update strategy rules
 *   - Get configuration by user ID
 *   - Update individual rule configurations
 * - Strategy Management:
 *   - Activate/deactivate strategies
 *   - Copy strategies
 *   - Generate unique strategy IDs
 *
 * Data Structure:
 * - `configuration-overview`: Strategy metadata (name, status, dates, etc.)
 * - `configurations`: Strategy rules (maxDailyTrades, riskReward, etc.)
 * - Strategies are linked by `configurationId` field
 *
 * Relations:
 * - Used by StrategyService for strategy operations
 * - Used by StrategyComponent for strategy management
 * - Used by EditStrategyComponent for rule updates
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class StrategyOperationsService {
```

```
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  // ===== CONFIGURATION-OVERVIEW (colección de metadatos) =====

  /**
   * Crear configuration-overview (solo metadatos)
   */
  async createConfigurationOverview(userId: string, name: string): Promise<string> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return '';
    }

    const overviewId = this.generateOverviewId();
    const now = Timestamp.now();

    const configurationOverview: ConfigurationOverview = {
      userId: userId,
      name: name,
      status: false, // Inicialmente inactiva
      created_at: now,
      updated_at: now,
      days_active: 0,
      configurationId: '' // Se establecerá después de crear la configuración
    };

    await setDoc(doc(this.db, 'configuration-overview', overviewId), configurationOverview);
    return overviewId;
  }

  /**
   * Obtener configuration-overview por ID (solo metadatos)
   */
  async getConfigurationOverview(overviewId: string): Promise<ConfigurationOverview | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const docRef = doc(this.db, 'configuration-overview', overviewId);
      const docSnap = await getDoc(docRef);

      if (docSnap.exists()) {
        const data = docSnap.data() as ConfigurationOverview;
        (data as any).id = docSnap.id; // Agregar ID del documento
        return data;
      }
      return null;
    } catch (error) {
      console.error('Error getting configuration overview:', error);
      return null;
    }
  }

  /**
   * Actualizar configuration-overview
   */
  async updateConfigurationOverview(overviewId: string, updates:
Partial<ConfigurationOverview>): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
```

```
      return;
    }

    const docRef = doc(this.db, 'configuration-overview', overviewId);
    const updateData = {
      ...updates,
      updated_at: Timestamp.now()
    };

    await updateDoc(docRef, updateData);
  }

  /**
   * Eliminar configuration-overview
   */
  async deleteConfigurationOverview(overviewId: string): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    const docRef = doc(this.db, 'configuration-overview', overviewId);
    await deleteDoc(docRef);
  }

  // ===== CONFIGURATIONS (colección de reglas) =====

  /**
   * Crear configuración (solo reglas + IDs)
   */
  async createConfiguration(userId: string, configurationOverviewId: string, configuration:
StrategyState): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    const configData = {
      ...configuration,
      configurationOverviewId: configurationOverviewId,
      userId: userId
    };

    await setDoc(doc(this.db, 'configurations', userId), configData);
  }

  /**
   * Obtener configuración por userId
   */
  async getConfiguration(userId: string): Promise<StrategyState | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const docRef = doc(this.db, 'configurations', userId);
      const docSnap = await getDoc(docRef);

      if (docSnap.exists()) {
        return docSnap.data() as StrategyState;
      }
      return null;
    } catch (error) {
      console.error('Error getting configuration:', error);
      return null;
    }
  }

  /**
   * Actualizar configuración
```

```
   */
  async updateConfiguration(userId: string, configuration: StrategyState): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    await updateDoc(doc(this.db, 'configurations', userId), configuration as any);
  }

  /**
   * Crear solo configuration (sin userId ni configurationOverviewId)
   */
  async createConfigurationOnly(configuration: StrategyState): Promise<string> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      throw new Error('Firestore not available');
    }

    try {
      const docRef = await addDoc(collection(this.db, 'configurations'), configuration as
any); return docRef.id;
    } catch (error) {
      console.error('Error creating configuration:', error);
      throw error;
    }
  }

  /**
   * Crear configuration-overview con configurationId
   */
  async createConfigurationOverviewWithConfigId(userId: string, name: string,
configurationId: string, shouldBeActive: boolean = false): Promise<string> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      throw new Error('Firestore not available');
    }

    try {
      const now = new Date();
      const overviewData: ConfigurationOverview = {
        userId,
        name,
        status: shouldBeActive, // Activa solo si no hay otra activa
        created_at: now,
        updated_at: now,
        days_active: 0,
        configurationId,
        dateActive: [now.toISOString()],
      };

      const docRef = await addDoc(collection(this.db, 'configuration-overview'),
overviewData as any);
      return docRef.id;
    } catch (error) {
      console.error('Error creating configuration overview:', error);
      throw error;
    }
  }

  /**
   * Actualizar configuration por ID
   */
  async updateConfigurationById(configurationId: string, configuration: StrategyState):
Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      throw new Error('Firestore not available');
    }

    try {
      const docRef = doc(this.db, 'configurations', configurationId);
      await updateDoc(docRef, configuration as any);
```

```
    } catch (error) {
      console.error('Error updating configuration by ID:', error);
      throw error;
    }
  }

  /**
   * Obtener configuración por ID
   */
  async getConfigurationById(configurationId: string): Promise<StrategyState | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const docRef = doc(this.db, 'configurations', configurationId);
      const docSnap = await getDoc(docRef);
      if (docSnap.exists()) {
        return docSnap.data() as StrategyState;
      }
      return null;
    } catch (error) {
      console.error('Error getting configuration by ID:', error);
      return null;
    }
  }

  /**
   * Obtener configuración por configurationOverviewId (método legacy para compatibilidad)
   */
  async getConfigurationByOverviewId(overviewId: string): Promise<StrategyState | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      // Buscar en configurations donde configurationOverviewId coincida
      const q = query(
        collection(this.db, 'configurations'),
        where('configurationOverviewId', '==', overviewId),
        limit(1)
      );

      const querySnapshot = await getDocs(q);
      if (!querySnapshot.empty) {
        const doc = querySnapshot.docs[0];
        return doc.data() as StrategyState;
      }

      return null;
    } catch (error) {
      console.error('Error getting configuration by overview ID:', error);
      return null;
    }
  }

  /**
   * Obtener todas las estrategias de un usuario
   */
  async getUserStrategyViews(userId: string): Promise<ConfigurationOverview[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      // 1. Primero obtener todos los configuration-overview del usuario
      const overviewQuery = query(
```

```
        collection(this.db, 'configuration-overview'),
        where('userId', '==', userId)
      );

      const overviewSnapshot = await getDocs(overviewQuery);

      const strategies: ConfigurationOverview[] = [];

      overviewSnapshot.forEach((doc) => {
        const data = doc.data() as ConfigurationOverview;
        (data as any).id = doc.id; // Agregar ID del documento
        strategies.push(data);
      });

      // Filtrar estrategias que no estén marcadas como deleted
      // Mostrar solo las que:
      // 1. No tienen el campo 'deleted' (estrategias antiguas)
      // 2. Tienen 'deleted: false' (explícitamente no eliminadas)
      const activeStrategies = strategies.filter(strategy =>
        strategy.deleted === undefined || strategy.deleted === false
      );

      // Ordenar manualmente por updated_at descendente
      activeStrategies.sort((a, b) => {
        const dateA = a.updated_at.toDate();
        const dateB = b.updated_at.toDate();
        return dateB.getTime() - dateA.getTime();
      });

      return activeStrategies;
    } catch (error) {
      console.error(''L Error getting user strategies:', error);
      return [];
    }
  }

  /**
   * Obtener configuración activa (método legacy para compatibilidad)
   */
  async getActiveConfiguration(userId: string): Promise<ConfigurationOverview | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      // Buscar estrategia activa en configuration-overview
      const q = query(
        collection(this.db, 'configuration-overview'),
        where('userId', '==', userId),
        where('status', '==', true),
        limit(1)
      );

      const querySnapshot = await getDocs(q);
      if (!querySnapshot.empty) {
        const doc = querySnapshot.docs[0];
        const data = doc.data() as ConfigurationOverview;
        (data as any).id = doc.id;
        return data;
      }

      return null;
    } catch (error) {
      console.error('Error getting active configuration:', error);
      return null;
    }
  }

  /**
```

```
   * Activar una estrategia
   */
  async activateStrategyView(userId: string, strategyId: string): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      // 1. Desactivar todas las estrategias del usuario
      const q = query(
        collection(this.db, 'configuration-overview'),
        where('userId', '==', userId),
        where('status', '==', true)
      );

      const querySnapshot = await getDocs(q);
      const batch = [];

      querySnapshot.forEach((doc) => {
        batch.push(updateDoc(doc.ref, { status: false }));
      });

      // 2. Activar la estrategia seleccionada
      batch.push(updateDoc(doc(this.db, 'configuration-overview', strategyId), {
        status: true,
        updated_at: Timestamp.now()
      }));

      // Ejecutar todas las actualizaciones
      await Promise.all(batch);
    } catch (error) {
      console.error('Error activating strategy:', error);
      throw error;
    }
  }

  /**
   * Actualizar fechas de activación/desactivación de estrategias
   */
  async updateStrategyDates(userId: string, strategyId: string, dateActive?: Date,
dateInactive?: Date): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      const strategyRef = doc(this.db, 'configuration-overview', strategyId);
      const strategyDoc = await getDoc(strategyRef);

      if (!strategyDoc.exists()) {
        throw new Error('Strategy not found');
      }

      const currentData = strategyDoc.data();
      const updateData: any = {};

      // Solo actualizar si se proporciona un valor válido
      if (dateActive !== undefined && dateActive !== null) {
        const currentDateActive = currentData['dateActive'] || [];
        const newDateActive = [...currentDateActive, dateActive.toISOString()];
        updateData.dateActive = newDateActive;

        // Si se está activando, cambiar status a true
        updateData.status = true;
      }

      // Solo actualizar si se proporciona un valor válido
      if (dateInactive !== undefined && dateInactive !== null) {
        const currentDateInactive = currentData['dateInactive'] || [];
```

```
        const newDateInactive = [...currentDateInactive, dateInactive.toISOString()];
        updateData.dateInactive = newDateInactive;

        // Si se está desactivando, cambiar status a false
        updateData.status = false;
      }

      // Actualizar timestamp solo si hay cambios
      if (Object.keys(updateData).length > 0) {
        updateData.updated_at = Timestamp.now();
        await updateDoc(strategyRef, updateData);
      }
    } catch (error) {
      console.error('Error updating strategy dates:', error);
      throw error;
    }
  }

  /**
   * Eliminar una estrategia
   */
  async deleteStrategyView(strategyId: string): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      // 1. Obtener el configurationId del overview
      const strategy = await this.getConfigurationOverview(strategyId);
      if (!strategy) {
        throw new Error('Strategy not found');
      }

      // 2. Eliminar configuration-overview
      await this.deleteConfigurationOverview(strategyId);

      // 3. Eliminar configuration usando configurationId
      if (strategy.configurationId) {
        try {
          const configDocRef = doc(this.db, 'configurations', strategy.configurationId);
          await deleteDoc(configDocRef);
        } catch (error) {
          console.warn('Configuration not found for deletion:', error);
        }
      }
    } catch (error) {
      console.error('Error deleting strategy:', error);
      throw error;
    }
  }

  /**
   * Marcar una estrategia como deleted (soft delete)
   */
  async markStrategyAsDeleted(strategyId: string): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      // 1. Obtener el configurationId del overview
      const strategy = await this.getConfigurationOverview(strategyId);
      if (!strategy) {
        throw new Error('Strategy not found');
      }

      const currentTimestamp = new Date();

```

```
      // 2. Marcar configuration-overview como deleted y agregar dateInactive
      const overviewDocRef = doc(this.db, 'configuration-overview', strategyId);
      const overviewDoc = await getDoc(overviewDocRef);

      if (overviewDoc.exists()) {
        const currentData = overviewDoc.data();
        const updateData: any = {
          deleted: true,
          deleted_at: Timestamp.now(),
          updated_at: Timestamp.now(),
          status: false // Marcar como inactiva
        };

        // Agregar dateInactive si la estrategia estaba activa
        if (currentData['status'] === true) {
          const currentDateInactive = currentData['dateInactive'] || [];
          const newDateInactive = [...currentDateInactive,
Timestamp.fromDate(currentTimestamp)];
          updateData.dateInactive = newDateInactive;
        }

        await updateDoc(overviewDocRef, updateData);
      }

      // 3. Marcar configuration como deleted usando configurationId
      if (strategy.configurationId) {
        try {
          const configDocRef = doc(this.db, 'configurations', strategy.configurationId);
          await updateDoc(configDocRef, {
            deleted: true,
            deleted_at: Timestamp.now(),
            updated_at: Timestamp.now()
          });
        } catch (error) {
          console.warn('Configuration not found for soft deletion:', error);
        }
      }
    } catch (error) {
      console.error('Error marking strategy as deleted:', error);
      throw error;
    }
  }

  /**
   * Obtener el número total de estrategias de un usuario (solo no eliminadas)
   */
  async getAllLengthConfigurationsOverview(userId: string): Promise<number> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return 0;
    }

    try {
      // Obtener todas las estrategias del usuario
      const overviewQuery = query(
        collection(this.db, 'configuration-overview'),
        where('userId', '==', userId)
      );

      const overviewSnapshot = await getDocs(overviewQuery);

      let count = 0;
      overviewSnapshot.forEach((doc) => {
        const data = doc.data() as ConfigurationOverview;
        // Solo contar las que no estén marcadas como deleted (deleted !== true)
        // Las que tienen deleted === false o no tienen el campo deleted se cuentan
        if (data.deleted === undefined || data.deleted === false) {
          count++;
        }
      });

```

```
      return count;
    } catch (error) {
      console.error('Error getting strategies count:', error);
      return 0;
    }
  }

  /**
   * Generar ID único para configuration-overview
   */
  private generateOverviewId(): string {
    return 'overview_' + Date.now() + '_' + Math.random().toString(36).substr(2, 9);
  }
}

```

Ø=ÜÄ shared\services\subscription-service.ts

```

import { Injectable } from '@angular/core';
import {
  collection,
  doc,
  getDocs,
  getDoc,
  addDoc,
  updateDoc,
  deleteDoc,
  query,
  orderBy,
  Timestamp,
  onSnapshot,
  limit
} from 'firebase/firestore';
import { db } from '../../firebase/firebase.init';
import { UserStatus } from '../../features/overview/models/overview';

/**
 * Interface for user subscription data.
 *
 * @interface Subscription
 */
export interface Subscription {
  id?: string;
  planId: string;
  status: UserStatus;
  created_at: Timestamp;
  updated_at: Timestamp;
  userId: string;
  transactionId?: string;
  periodStart?: Timestamp;
  periodEnd?: Timestamp;
  cancelAtPeriodEnd?: boolean;
}

/**
 * Service for managing user subscriptions in Firebase.
 *
 * This service provides CRUD operations for user subscriptions, including
 * creating, reading, updating, and listening to subscription changes.
 * It manages subscription data in the Firestore subcollection
 * `users/{userId}/subscription`.
 *
 * Features:
 * - Get user's latest subscription
 * - Listen to subscription changes in real-time
```

```
 * - Create new subscriptions
 * - Update existing subscriptions
 * - Delete subscriptions
 * - Get subscription by ID
 * - Get all user subscriptions
 *
 * Subscription Structure:
 * - Stored in: `users/{userId}/subscription/{subscriptionId}`
 * - Ordered by `created_at` descending
 * - Only one active subscription per user expected
 *
 * Relations:
 * - Used by AuthService for plan management
 * - Used by PlanSettingsComponent for subscription display
 * - Used by PlanLimitationsGuard for plan validation
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class SubscriptionService {
  constructor() {}

  /**
   * Obtiene la última suscripción de un usuario (único documento esperado)
   * @param userId ID del usuario
   * @returns Promise con la suscripción o null si no existe
   */
  async getUserLatestSubscription(userId: string): Promise<Subscription | null> {
    try {
      const paymentsRef = collection(db, 'users', userId, 'subscription');
      const q = query(paymentsRef, orderBy('created_at', 'desc'), limit(1));
      const querySnapshot = await getDocs(q);

      if (querySnapshot.empty) {
        return null;
      }

      const latestDoc = querySnapshot.docs[0];
      const data = latestDoc.data();
      return data as unknown as Subscription;
    } catch (error) {
      console.error('L Error al obtener suscripción del usuario:', error);
      throw error;
    }
  }

  /**
   * Escucha cambios en la última suscripción del usuario (único documento esperado)
   * Devuelve una función para desuscribirse
   */
  listenToUserLatestSubscription(
    userId: string,
    handler: (subscription: Subscription | null) => void
  ): () => void {
    const paymentsRef = collection(db, 'users', userId, 'subscription');
    const q = query(paymentsRef, orderBy('created_at', 'desc'), limit(1));
    const unsubscribe = onSnapshot(q, (snapshot) => {
      if (snapshot.empty) {
        handler(null);
        return;
      }
      const latestDoc = snapshot.docs[0];
      const data = latestDoc.data();
      handler({ id: latestDoc.id, ...data } as Subscription);
    }, (error) => {
      console.error('L Error en listener de suscripción:', error);
```

```
      handler(null);
    });
    return unsubscribe;
  }

  // TODO: IMPLEMENTAR ENDPOINT DE VERIFICACIÓN DE PAGO - Reemplazar Firebase con API real
  /**
   * Obtiene un pago específico por ID
   * @param userId ID del usuario
   * @param paymentId ID del pago
   * @returns Promise con el pago o null si no existe
   */
  async getSubscriptionById(userId: string, paymentId: string): Promise<Subscription | null>
{   try {
      const paymentRef = doc(db, 'users', userId, 'subscription', paymentId);
      const paymentSnap = await getDoc(paymentRef);

      if (paymentSnap.exists()) {
        return {
          id: paymentSnap.id,
          ...paymentSnap.data()
        } as Subscription;
      } else {
        return null;
      }
    } catch (error) {
      console.error('Error al obtener pago:', error);
      throw error;
    }
  }

  // TODO: IMPLEMENTAR ENDPOINT DE CREACIÓN DE PAGO - Reemplazar Firebase con API real
  /**
   * Crea un nuevo pago
   * @param userId ID del usuario
   * @param paymentData Datos del pago (sin id)
   * @returns Promise con el ID del pago creado
   */
  async createSubscription(userId: string, paymentData: Omit<Subscription, 'id' |
'created_at' | 'updated_at'>): Promise<string> {
    try {
      const paymentsRef = collection(db, 'users', userId, 'subscription');
      const now = Timestamp.now();

      const newPayment = {
        ...paymentData,
        created_at: now,
        updated_at: now
      };

      const docRef = await addDoc(paymentsRef, newPayment);
      return docRef.id;
    } catch (error) {
      console.error('Error al crear pago:', error);
      throw error;
    }
  }

  /**
   * Actualiza un pago existente
   * @param userId ID del usuario
   * @param paymentId ID del pago
   * @param updateData Datos a actualizar
   * @returns Promise void
   */
  async updateSubscription(userId: string, paymentId: string, updateData:
Partial<Omit<Subscription, 'id' | 'created_at' | 'userId'>>): Promise<void> {
    try {
      const paymentRef = doc(db, 'users', userId, 'subscription', paymentId);
      const updatePayload = {
        ...updateData,
        updated_at: Timestamp.now()
```

```
      };

      await updateDoc(paymentRef, updatePayload);
    } catch (error) {
      console.error('Error al actualizar pago:', error);
      throw error;
    }
  }

  /**
   * Elimina un pago
   * @param userId ID del usuario
   * @param paymentId ID del pago
   * @returns Promise void
   */
  async deleteSubscription(userId: string, paymentId: string): Promise<void> {
    try {
      const paymentRef = doc(db, 'users', userId, 'subscription', paymentId);
      await deleteDoc(paymentRef);
    } catch (error) {
      console.error('Error al eliminar pago:', error);
      throw error;
    }
  }

  /**
   * Obtiene pagos filtrados por estado
   * @param userId ID del usuario
   * @param status Estado del pago
   * @returns Promise con array de pagos filtrados
   */
  async getSubscriptionsByStatus(userId: string, status: Subscription['status']):
Promise<Subscription[]> {
    try {
      const paymentsRef = collection(db, 'users', userId, 'subscription');
      const q = query(
        paymentsRef,
        orderBy('created_at', 'desc')
      );
      const querySnapshot = await getDocs(q);

      return querySnapshot.docs
        .map(doc => ({
          id: doc.id,
          ...doc.data()
        } as Subscription))
        .filter(payment => payment.status === status);
    } catch (error) {
      console.error('Error al obtener pagos por estado:', error);
      throw error;
    }
  }

  /**
   * Obtiene el total de pagos de un usuario
   * @param userId ID del usuario
   * @returns Promise con el número total de pagos
   */
  async getTotalSubscriptionsCount(userId: string): Promise<number> {
    try {
      const paymentsRef = collection(db, 'users', userId, 'subscription');
      const querySnapshot = await getDocs(paymentsRef);
      return querySnapshot.size;
    } catch (error) {
      console.error('Error al obtener conteo de pagos:', error);
      throw error;
    }
  }

  /**
   * Método de debug para verificar la estructura de la base de datos
```

```
   * @param userId ID del usuario
   */
}
```

Ø=ÜÄ shared\services\timezone.service.ts

```
import { Injectable } from '@angular/core';
import * as moment from 'moment-timezone';

@Injectable({
  providedIn: 'root'
})
export class TimezoneService {

  constructor() { }

  /**
   * Obtener la zona horaria del usuario desde el navegador
   */
  getUserTimezone(): string {
    try {
      return Intl.DateTimeFormat().resolvedOptions().timeZone;
    } catch (error) {
      console.warn('No se pudo detectar la zona horaria del usuario, usando UTC');
      return 'UTC';
    }
  }

  /**
   * Convertir una fecha local del usuario a UTC
   * MEJORA: Manejo correcto de fechas que ya están en UTC
   * @param localDate Fecha en zona horaria local del usuario
   * @param userTimezone Zona horaria del usuario (opcional, se detecta automáticamente)
   * @returns Fecha en UTC
   */
  convertToUTC(localDate: Date | string, userTimezone?: string): Date {
    const timezone = userTimezone || this.getUserTimezone();

    try {
      // Si la fecha ya es un objeto Date, verificar si ya está en UTC
      if (localDate instanceof Date) {
        // Verificar si la fecha ya está en UTC (sin offset de zona horaria)
        const dateString = localDate.toISOString();
        if (dateString.includes('Z') || dateString.includes('+00:00')) {
          // Ya está en UTC, devolver tal como está
          return localDate;
        }
      }

      // Si es string, verificar si ya tiene indicador UTC
      if (typeof localDate === 'string') {
        if (localDate.includes('Z') || localDate.includes('+00:00') ||
localDate.includes('UTC')) {
          // Ya está en UTC, convertir directamente
          return new Date(localDate);
        }
      }

      // Crear momento en la zona horaria del usuario
      const momentInUserTz = moment.tz(localDate, timezone);

      // Convertir a UTC
      const utcMoment = momentInUserTz.utc();

      return utcMoment.toDate();
    } catch (error) {
      console.error('Error convirtiendo fecha a UTC:', error);
```

```
      // Fallback: usar la fecha original
      return new Date(localDate);
    }
  }

  /**
   * Convertir una fecha UTC a la zona horaria del usuario
   * @param utcDate Fecha en UTC
   * @param userTimezone Zona horaria del usuario (opcional, se detecta automáticamente)
   * @returns Fecha en zona horaria del usuario
   */
  convertFromUTC(utcDate: Date | string, userTimezone?: string): Date {
    const timezone = userTimezone || this.getUserTimezone();

    try {
      // Crear momento UTC
      const utcMoment = moment.utc(utcDate);

      // Convertir a la zona horaria del usuario
      const localMoment = utcMoment.tz(timezone);

      return localMoment.toDate();
    } catch (error) {
      console.error('Error convirtiendo fecha desde UTC:', error);
      // Fallback: usar la fecha original
      return new Date(utcDate);
    }
  }

  /**
   * Verificar si una fecha está dentro de un rango de horas específico
   * @param date Fecha a verificar
   * @param startTime Hora de inicio (formato HH:mm)
   * @param endTime Hora de fin (formato HH:mm)
   * @param timezone Zona horaria para la comparación
   * @returns true si la fecha está dentro del rango
   */
  isWithinTimeRange(
    date: Date | string,
    startTime: string,
    endTime: string,
    timezone: string
  ): boolean {
    try {
      const momentDate = moment.tz(date, timezone);
      const currentTime = momentDate.format('HH:mm');

      // Si el rango cruza medianoche (ej: 22:00 - 06:00)
      if (startTime > endTime) {
        return currentTime >= startTime || currentTime <= endTime;
      } else {
        // Rango normal (ej: 09:00 - 17:00)
        return currentTime >= startTime && currentTime <= endTime;
      }
    } catch (error) {
      console.error('Error verificando rango de tiempo:', error);
      return false;
    }
  }

  /**
   * Obtener la fecha actual en UTC
   * @returns Fecha actual en UTC
   */
  getCurrentUTC(): Date {
    return moment.utc().toDate();
  }

  /**
   * Obtener la fecha actual en la zona horaria del usuario
```

```
   * @param userTimezone Zona horaria del usuario (opcional)
   * @returns Fecha actual en zona horaria del usuario
   */
  getCurrentLocal(userTimezone?: string): Date {
    const timezone = userTimezone || this.getUserTimezone();
    return moment.tz(timezone).toDate();
  }

  /**
   * Formatear fecha para mostrar en la zona horaria del usuario
   * @param date Fecha a formatear
   * @param format Formato deseado (por defecto: 'YYYY-MM-DD HH:mm:ss')
   * @param userTimezone Zona horaria del usuario (opcional)
   * @returns Fecha formateada
   */
  formatDate(
    date: Date | string,
    format: string = 'YYYY-MM-DD HH:mm:ss',
    userTimezone?: string
  ): string {
    const timezone = userTimezone || this.getUserTimezone();

    try {
      return moment.tz(date, timezone).format(format);
    } catch (error) {
      console.error('Error formateando fecha:', error);
      return new Date(date).toString();
    }
  }

  /**
   * Comparar dos fechas considerando zona horaria
   * @param date1 Primera fecha
   * @param date2 Segunda fecha
   * @param timezone Zona horaria para la comparación
   * @returns -1 si date1 < date2, 0 si son iguales, 1 si date1 > date2
   */
  compareDates(
    date1: Date | string,
    date2: Date | string,
    timezone?: string
  ): number {
    const tz = timezone || this.getUserTimezone();

    try {
      const moment1 = moment.tz(date1, tz);
      const moment2 = moment.tz(date2, tz);

      if (moment1.isBefore(moment2)) return -1;
      if (moment1.isAfter(moment2)) return 1;
      return 0;
    } catch (error) {
      console.error('Error comparando fechas:', error);
      return 0;
    }
  }

  /**
   * Verificar si una fecha está en el día de la semana especificado
   * @param date Fecha a verificar
   * @param dayOfWeek Día de la semana (0=domingo, 1=lunes, ..., 6=sábado)
   * @param timezone Zona horaria para la verificación
   * @returns true si la fecha está en el día especificado
   */
  isDayOfWeek(
    date: Date | string,
    dayOfWeek: number,
    timezone?: string
  ): boolean {
    const tz = timezone || this.getUserTimezone();
```

```
201
202       try {
203         const momentDate = moment.tz(date, tz);
204         return momentDate.day() === dayOfWeek;
205       } catch (error) {
206         console.error('Error verificando día de la semana:', error);
207         return false;
208       }
209     }
210
211     /**
212      * Convertir fecha de trade del servidor a UTC
213      * MÉTODO ESPECÍFICO: Para fechas que vienen del servidor y pueden estar en diferentes
214 formatos
215      * @param tradeDate Fecha del trade (puede ser timestamp, string, o Date)
216      * @returns Fecha en UTC
217      */
218     convertTradeDateToUTC(tradeDate: any): Date {
219       try {
220         let utcDate: Date;
221
222         // Si es un timestamp numérico (milisegundos)
223         if (typeof tradeDate === 'number') {
224           utcDate = new Date(tradeDate);
225           // Verificar si es válido
226           if (isNaN(utcDate.getTime())) {
227             throw new Error('Timestamp inválido');
228           }
229         }
230         // Si es un string
231         else if (typeof tradeDate === 'string') {
232           // Si contiene 'Z' o '+00:00', ya está en UTC
233           if (tradeDate.includes('Z') || tradeDate.includes('+00:00')) {
234             utcDate = new Date(tradeDate);
235           }
236           // Si es un timestamp en string
237           else {
238             const numericValue = parseInt(tradeDate);
239             if (!isNaN(numericValue)) {
240               utcDate = new Date(numericValue);
241             } else {
242               utcDate = new Date(tradeDate);
243             }
244           }
245         }
246         // Si ya es un Date
247         else if (tradeDate instanceof Date) {
248           utcDate = tradeDate;
249         }
250         // Fallback
251         else {
252           utcDate = new Date(tradeDate);
253         }
254
255         // FORZAR UTC: Crear una nueva fecha usando UTC para asegurar que esté en UTC
256         const utcTimestamp = Date.UTC(
257           utcDate.getUTCFullYear(),
258           utcDate.getUTCMonth(),
259           utcDate.getUTCDate(),
260           utcDate.getUTCHours(),
261           utcDate.getUTCMinutes(),
262           utcDate.getUTCSeconds(),
263           utcDate.getUTCMilliseconds()
264         );
265
266         return new Date(utcTimestamp);
267       } catch (error) {
268         console.error('Error convirtiendo fecha de trade a UTC:', error);
269         return new Date();
270       }
```

```
271
272   /**
273    * Obtener información de debug sobre zona horaria
274    * @param date Fecha a analizar
275    * @param userTimezone Zona horaria del usuario (opcional)
276    * @returns Información de debug
277    */
278   getTimezoneDebugInfo(date: Date | string, userTimezone?: string): any {
279     const timezone = userTimezone || this.getUserTimezone();
280
281     try {
282       const momentDate = moment.tz(date, timezone);
283       const utcDate = moment.utc(date);
284
285       return {
286         originalDate: date,
287         userTimezone: timezone,
288         localTime: momentDate.format('YYYY-MM-DD HH:mm:ss'),
289         utcTime: utcDate.format('YYYY-MM-DD HH:mm:ss'),
290         offset: momentDate.utcOffset(),
291         offsetString: momentDate.format('Z'),
292         dayOfWeek: momentDate.day(),
293         dayName: momentDate.format('dddd')
294       };
295     } catch (error) {
296       return {
297         error: error instanceof Error ? error.message : 'Unknown error',
298         originalDate: date,
299         userTimezone: timezone
300       };
301     }
302   }
303 }
304
```

Ø=ÜÄ shared\services\tokens-operations.service.ts

```
1  import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
2  import { getFirestore, doc, setDoc, deleteDoc } from 'firebase/firestore';
3  import { isPlatformBrowser } from '@angular/common';
4
5  /**
6   * Interface for link token data.
7   *
8   * @interface LinkToken
9   */
10 export interface LinkToken {
11   id: string;
12   [key: string]: any;
13 }
14
15 /**
16  * Service for managing link tokens in Firebase.
17  *
18  * This service provides operations for creating and deleting link tokens
19  * that are used for user authentication and account linking. Tokens are
20  * stored in the `tokens` collection.
21  *
22  * Features:
23  * - Create link token
24  * - Delete link token
25  *
26  * Token Structure:
27  * - Stored in: `tokens/{tokenId}`
28  * - Used for: User authentication, account linking
29  *
```

```
 * Relations:
 * - Used by AuthService for token management
 * - Used for logout everywhere functionality (token revocation)
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class TokensOperationsService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Crear token de enlace
   */
  async createLinkToken(token: LinkToken): Promise<void> {
    if (this.db) {
      await setDoc(doc(this.db, 'tokens', token.id), token);
    } else {
      console.warn('Firestore not available in SSR');
      return;
    }
  }

  /**
   * Eliminar token de enlace
   */
  async deleteLinkToken(tokenId: string): Promise<void> {
    if (this.db) {
      await deleteDoc(doc(this.db, 'tokens', tokenId));
    } else {
      console.warn('Firestore not available in SSR');
      return;
    }
  }
}

```

Ø=ÜÄ shared\services\tradelocker-api.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

/**
 * Interface for TradeLocker API credentials.
 *
 * @interface TradeLockerCredentials
 */
export interface TradeLockerCredentials {
  email: string;
  password: string;
  server: string;
}
```

```
16
17  /**
18   * Interface for TradeLocker token response.
19   *
20   * @interface TradeLockerTokenResponse
21   */
22  export interface TradeLockerTokenResponse {
23    accessToken: string;
24    tokenType: string;
25    expiresIn: number;
26  }
27
28  /**
29   * Interface for TradeLocker account data.
30   *
31   * @interface TradeLockerAccount
32   */
33  export interface TradeLockerAccount {
34    accountId: string;
35    accountName: string;
36    balance: number;
37    currency: string;
38    server: string;
39  }
40
41  /**
42   * Service for interacting with the TradeLocker API.
43   *
44   * This service provides methods to authenticate with TradeLocker, fetch
45   * account balances, trading history, and instrument details. It handles
46   * JWT token management and API communication.
47   *
48   * Features:
49   * - JWT token authentication
50   * - Token refresh
51   * - Account validation
52   * - Account balance fetching
53   * - Trading history retrieval
54   * - Instrument details fetching
55   * - All instruments listing
56   * - User key generation
57   *
58   * API Endpoints:
59   * - Base URL: https://demo.tradelocker.com/backend-api
60   * - Auth: /auth/jwt/token, /auth/jwt/refresh
61   * - Trade: /trade/accounts/{accountId}/state, /trade/accounts/{accountId}/ordersHistory
62   * - Instruments: /trade/instruments/{tradableInstrumentId}, /trade/accounts/{accountId}/
63  instruments
64   * Relations:
65   * - Used by ReportService for fetching trading data
66   * - Used by CreateAccountPopupComponent for account validation
67   * - Used by TradingAccountsComponent for balance fetching
68   *
69   * @service
70   * @injectable
71   * @providedIn root
72   */
73  @Injectable({
74    providedIn: 'root'
75  })
76  export class TradeLockerApiService {
77    private readonly baseUrl = 'https://demo.tradelocker.com/backend-api';
78
79    constructor(private http: HttpClient) {}
80
81    /**
82     * Get JWT token from TradeLocker
83     */
84    getJWTToken(credentials: TradeLockerCredentials): Observable<TradeLockerTokenResponse> {
85      const tokenUrl = `${this.baseUrl}/auth/jwt/token`;
```

```
86
87      const headers = new HttpHeaders({
88        'Content-Type': 'application/json',
89      });
90
91      const body = {
92        email: credentials.email,
93        password: credentials.password,
94        server: credentials.server
95      };
96
97      return this.http.post<TradeLockerTokenResponse>(tokenUrl, body, { headers });
98    }
99
100   refreshToken(accessToken: string): Observable<any> {
101     const refreshUrl = `${this.baseUrl}/auth/jwt/refresh`;
102
103     const headers = new HttpHeaders({
104       'Content-Type': 'application/json',
105       'Authorization': `Bearer ${accessToken}`
106     });
107
108     return this.http.post<any>(refreshUrl, { headers });
109   }
110
111   /**
112    * Validate account credentials in TradeLocker
113    */
114   async validateAccount(credentials: TradeLockerCredentials): Promise<boolean> {
115     try {
116       const tokenResponse = await this.getJWTToken(credentials).toPromise();
117       return !!(tokenResponse && tokenResponse.accessToken);
118     } catch (error) {
119       console.error('Error validating account in TradeLocker:', error);
120       return false;
121     }
122   }
123
124   /**
125    * Get account balance from TradeLocker
126    */
127   getAccountBalance(accountId: string, userKey: string, accountNumber: number):
Observable<any> {
128     const balanceUrl = `${this.baseUrl}/trade/accounts/${accountId}/state`;
129
130     const headers = new HttpHeaders({
131       'Content-Type': 'application/json',
132       'Authorization': `Bearer ${userKey}`,
133       'accNum': accountNumber.toString()
134     });
135
136     return this.http.get(balanceUrl, { headers });
137   }
138
139   /**
140    * Get trading history from TradeLocker
141    */
142   getTradingHistory(userKey: string, accountId: string, accNum: number): Observable<any> {
143     const historyUrl = `${this.baseUrl}/trade/accounts/${accountId}/ordersHistory`;
144
145     const headers = new HttpHeaders({
146       'Content-Type': 'application/json',
147       'Authorization': `Bearer ${userKey}`,
148       'accNum': accNum.toString()
149     });
150
151     return this.http.get(historyUrl, { headers });
152   }
153
154   /**
155    * Get user key for API calls
```

```
   */
  getUserKey(email: string, password: string, server: string): Observable<string> {
    const credentials: TradeLockerCredentials = { email, password, server };
    return this.getJWTToken(credentials).pipe(
      map(response => response.accessToken)
    );
  }

  /**
   * Get instrument details
   */
  getInstrumentDetails(accessToken: string, tradableInstrumentId: string, routeId: string,
accNum: number): Observable<any> {
    const instrumentsUrl = `${this.baseUrl}/trade/instruments/${tradableInstrumentId}`;

    const headers = new HttpHeaders({
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${accessToken}`,
      'accNum': accNum.toString()
    });

    const params = {
      routeId: routeId
    };

    return this.http.get(instrumentsUrl, { headers, params });
  }

  /**
   * Get all instruments for an account
   */
  getAllInstruments(accessToken: string, accountId: string, accNum: number): Observable<any>
{   const instrumentsUrl = `${this.baseUrl}/trade/accounts/${accountId}/instruments`;

    const headers = new HttpHeaders({
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${accessToken}`,
      'accNum': accNum.toString()
    });

    return this.http.get(instrumentsUrl, { headers });
  }

}

```

Ø=ÜÄ shared\services\user-management.service.ts

```
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { getFirestore, collection, getDocs, doc, getDoc, updateDoc, deleteDoc, query, where,
orderBy, limit } from 'firebase/firestore';
import { isPlatformBrowser } from '@angular/common';
import { User, UserStatus } from '../../features/overview/models/overview';

/**
 * Service for managing user data operations (administrative).
 *
 * This service provides administrative operations for user management,
 * including fetching all users, filtering by status, and getting top users.
 * It's designed for admin interfaces and user management dashboards.
 *
 * Features:
 * - Get all users from Firebase
 * - Get user by ID
 * - Update user data
 * - Delete user
 * - Get users by status (active, banned, etc.)
 * - Get top users (ordered by number of trades)
```

```
 *
 * Usage:
 * Primarily used by admin components like UsersDetailsComponent for
 * managing and viewing user data.
 *
 * Relations:
 * - Used by UsersDetailsComponent for user management
 * - Used by OverviewComponent for user statistics
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class UserManagementService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Get all users from Firebase
   */
  async getAllUsers(): Promise<User[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const snapshot = await getDocs(collection(this.db, 'users'));
      const users: User[] = [];

      snapshot.forEach((doc) => {
        const userData = doc.data() as User;
        (userData as any).id = doc.id;
        users.push(userData);
      });

      return users;
    } catch (error) {
      console.error('Error getting all users:', error);
      return [];
    }
  }

  /**
   * Get user by ID
   */
  async getUserById(userId: string): Promise<User | null> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return null;
    }

    try {
      const userDoc = await getDoc(doc(this.db, 'users', userId));
      if (userDoc.exists()) {
        const userData = userDoc.data() as User;
        (userData as any).id = userDoc.id;
        return userData;
      }
```

```
      return null;
    } catch (error) {
      console.error('Error getting user by ID:', error);
      return null;
    }
  }

  /**
   * Update user data
   */
  async updateUser(userId: string, userData: Partial<User>): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      await updateDoc(doc(this.db, 'users', userId), userData);
    } catch (error) {
      console.error('Error updating user:', error);
      throw error;
    }
  }

  /**
   * Delete user
   */
  async deleteUser(userId: string): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }

    try {
      await deleteDoc(doc(this.db, 'users', userId));
    } catch (error) {
      console.error('Error deleting user:', error);
      throw error;
    }
  }

  /**
   * Get users by status
   */
  async getUsersByStatus(status: UserStatus): Promise<User[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const q = query(
        collection(this.db, 'users'),
        where('status', '==', status)
      );

      const snapshot = await getDocs(q);
      const users: User[] = [];

      snapshot.forEach((doc) => {
        const userData = doc.data() as User;
        (userData as any).id = doc.id;
        users.push(userData);
      });

      return users;
    } catch (error) {
      console.error('Error getting users by status:', error);
      return [];
    }
```

```
  }

  /**
   * Get top users (ordered by some criteria)
   */
  async getTopUsers(limitCount: number = 10): Promise<User[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const q = query(
        collection(this.db, 'users'),
        orderBy('number_trades', 'desc'),
        limit(limitCount)
      );

      const snapshot = await getDocs(q);
      const users: User[] = [];

      snapshot.forEach((doc) => {
        const userData = doc.data() as User;
        (userData as any).id = doc.id;
        users.push(userData);
      });

      return users;
    } catch (error) {
      console.error('Error getting top users:', error);
      return [];
    }
  }
}

```

Ø=ÜÄ shared\services\users-operations.service.ts

```
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';
import { getFirestore, doc, setDoc, getDoc, collection, getDocs, deleteDoc } from 'firebase/
firestore';
import { isPlatformBrowser } from '@angular/common';
import { User } from '../../features/overview/models/overview';

/**
 * Service for user data operations in Firebase.
 *
 * This service provides CRUD operations for user documents in Firestore.
 * It handles user creation, retrieval, updates, and deletion. It's used
 * throughout the application for user data management.
 *
 * Features:
 * - Get user data by UID
 * - Create new user
 * - Get user by ID
 * - Get user by email
 * - Update user data
 * - Get all users
 * - Delete user
 *
 * User Data Structure:
 * - Stored in: `users/{userId}`
 * - Includes: profile data, trading statistics, subscription info
 *
 * Relations:
 * - Used by AuthService for user operations
 * - Used by ProfileDetailsComponent for profile updates
```

```
 * - Used by various components for user data access
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class UsersOperationsService {
  private isBrowser: boolean;
  private db: ReturnType<typeof getFirestore> | null = null;

  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    this.isBrowser = isPlatformBrowser(this.platformId);
    if (this.isBrowser) {
      const { firebaseApp } = require('../../firebase/firebase.init.ts');
      this.db = getFirestore(firebaseApp);
    }
  }

  /**
   * Obtener datos de un usuario por UID
   */
  async getUserData(uid: string): Promise<User> {
    if (this.db) {
      const userDoc = doc(this.db, 'users', uid);
      return getDoc(userDoc).then((doc) => {
        if (doc.exists()) {
          return doc.data() as User;
        } else {
          throw new Error('User not found');
        }
      });
    } else {
      console.warn('Firestore not available in SSR');
      return Promise.resolve({} as User);
    }
  }

  /**
   * Crear usuario
   */
  async createUser(user: User): Promise<void> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return;
    }
    await setDoc(doc(this.db, 'users', user.id), user);
  }

  /**
   * Obtener un usuario por su ID
   */
  async getUserById(userId: string): Promise<User | null> {
    try {
      if (!this.isBrowser || !this.db) {
        return null;
      }

      const userDoc = await getDoc(doc(this.db, 'users', userId));
      if (userDoc.exists()) {
        return { id: userDoc.id, ...userDoc.data() } as User;
      }
      return null;
    } catch (error) {
      console.error('Error obteniendo usuario por ID:', error);
      return null;
    }
  }
```

```
  /**
   * Buscar un usuario por su email
   */
  async getUserByEmail(email: string): Promise<User | null> {
    try {
      if (!this.isBrowser || !this.db) {
        return null;
      }

      const usersSnapshot = await getDocs(collection(this.db, 'users'));

      for (const doc of usersSnapshot.docs) {
        const userData = doc.data() as User;
        if (userData.email === email) {
          return { ...userData, id: doc.id } as User;
        }
      }

      return null;
    } catch (error) {
      console.error('Error buscando usuario por email:', error);
      return null;
    }
  }

  /**
   * Actualizar un usuario existente
   */
  async updateUser(userId: string, userData: Partial<User>): Promise<void> {
    try {
      if (!this.isBrowser || !this.db) {
        throw new Error('No se puede actualizar usuario en el servidor');
      }

      await setDoc(doc(this.db, 'users', userId), {
        ...userData,
        lastUpdated: new Date().getTime()
      }, { merge: true });
    } catch (error) {
      console.error('Error actualizando usuario:', error);
      throw error;
    }
  }

  /**
   * Obtener todos los usuarios
   */
  async getAllUsers(): Promise<User[]> {
    if (!this.db) {
      console.warn('Firestore not available in SSR');
      return [];
    }

    try {
      const snapshot = await getDocs(collection(this.db, 'users'));
      const users: User[] = [];

      snapshot.forEach((doc) => {
        const data = doc.data() as User;
        (data as any).id = doc.id;
        users.push(data);
      });

      return users;
    } catch (error) {
      console.error('Error getting all users:', error);
      return [];
    }
  }
```

```
169
170   /**
171    * Eliminar un usuario
172    */
173   async deleteUser(userId: string): Promise<void> {
174     try {
175       if (!this.isBrowser || !this.db) {
176         throw new Error('No se puede eliminar usuario en el servidor');
177       }
178
179       await deleteDoc(doc(this.db, 'users', userId));
180       console.log('Usuario eliminado exitosamente:', userId);
181     } catch (error) {
182       console.error('Error eliminando usuario:', error);
183       throw error;
184     }
185   }
186 }
187
```

Ø=ÜÁ shared\sidebar-menu

Ø=ÜÄ shared\sidebar-menu\sidebar.component.ts

```
1  import { Component, OnInit, OnDestroy } from '@angular/core';
2  import { Router, RouterLink, RouterLinkActive, NavigationEnd } from '@angular/router';
3  import { filter } from 'rxjs/operators';
4  import { AuthService } from '../../features/auth/service/authService';
5  import { Store } from '@ngrx/store';
6  import { selectUser } from '../../features/auth/store/user.selectios';
7  import { setUserData } from '../../features/auth/store/user.actions';
8  import { User, UserStatus } from '../../features/overview/models/overview';
9
10 /**
11  * Sidebar navigation component for the application.
12  *
13  * This component provides the main navigation sidebar with user information,
14  * menu items, and logout functionality. It dynamically adjusts its width based
15  * on the current route and user state, and manages CSS custom properties for
16  * layout consistency.
17  *
18  * Features:
19  * - User information display (name, initials, admin status, ban status)
20  * - Collapsible sidebar (minimized/expanded states)
21  * - Dynamic width management via CSS custom properties
22  * - Route-based visibility (hidden on login/signup pages)
23  * - Logout functionality
24  * - Dashboard section toggle
25  * - Responsive to route changes
26  *
27  * Width Management:
28  * - Expanded: 230px
29  * - Minimized: 80px
30  * - Hidden: 0px (on login/signup routes)
31  * - Uses CSS custom property: --sidebar-width
32  *
33  * Relations:
34  * - AuthService: Handles logout functionality
35  * - Store (NgRx): Gets current user data
36  * - Router: Monitors route changes for width adjustment
37  *
38  * @component
39  * @selector app-sidebar
```

```
 * @standalone true
 */
@Component({
  selector: 'app-sidebar',
  imports: [RouterLink, RouterLinkActive],
  templateUrl: './sidebar.component.html',
  styleUrl: './sidebar.component.scss',
  standalone: true,
})
export class Sidebar implements OnDestroy {
  isDashboardOpen = true;
  isSidebarMinimized = false;
  userName: string = '';
  lastName: string = '';
  isAdmin: boolean = false;
  userToken: string = '';
  isBanned: boolean = false;
  private hasInitializedWidth = false;

  constructor(
    private authService: AuthService,
    private router: Router,
    private store: Store
  ) {
    // Inicializar el ancho a 0 inmediatamente para evitar el flash
    if (typeof document !== 'undefined') {
      const root = document.documentElement;
      root.style.setProperty('--sidebar-width', '0px');
    }

    this.store.select(selectUser).subscribe((user) => {
      this.userName = user?.user?.firstName || '';
      this.lastName = user?.user?.lastName || '';
      this.isAdmin = user?.user?.isAdmin || false;
      this.userToken = user?.user?.tokenId || '';
      this.isBanned = user?.user?.status === UserStatus.BANNED;

      // Solo actualizar el ancho del sidebar una vez que tengamos datos del usuario
      if (this.userName && !this.hasInitializedWidth) {
        this.hasInitializedWidth = true;
        this.updateSidebarWidth(this.router.url);
      } else if (!this.userName && this.hasInitializedWidth) {
        // Si el usuario se desloguea, resetear
        this.hasInitializedWidth = false;
        if (typeof document !== 'undefined') {
          const root = document.documentElement;
          root.style.setProperty('--sidebar-width', '0px');
        }
      }
    });

    // Escuchar cambios de ruta para ajustar el sidebar
    this.router.events
      .pipe(filter(event => event instanceof NavigationEnd))
      .subscribe((event: NavigationEnd) => {
        // Solo actualizar si ya tenemos usuario
        if (this.userName) {
          this.updateSidebarWidth(event.url);
        }
      });
  }

  private updateSidebarWidth(url: string) {
    if (typeof document === 'undefined') return;

    const root = document.documentElement;
    const routesWithoutSidebar = ['/login', '/signup'];
    const hasSidebar = !routesWithoutSidebar.some(route => url.includes(route));

    if (hasSidebar) {
```

```
      // Hay sidebar, usar el ancho según el estado
      if (this.isSidebarMinimized) {
        root.style.setProperty('--sidebar-width', '80px');
      } else {
        root.style.setProperty('--sidebar-width', '230px');
      }
    } else {
      // No hay sidebar, resetear a 0
      root.style.setProperty('--sidebar-width', '0px');
    }
  }

  ngOnDestroy() {
    // Resetear la variable CSS cuando el componente se destruye (ej: al ir a login)
    if (typeof document !== 'undefined') {
      const root = document.documentElement;
      root.style.setProperty('--sidebar-width', '0px');
    }
  }

  onlyNameInitials() {
    return this.userName.charAt(0) + this.lastName.charAt(1);
  }

  toggleDashboard() {
    this.isDashboardOpen = !this.isDashboardOpen;
  }

  toggleSidebar() {
    this.isSidebarMinimized = !this.isSidebarMinimized;
    // Actualizar variable CSS usando la nueva lógica
    this.updateSidebarWidth(this.router.url);
  }

  logout() {
    this.authService
      .logout()
      .then(() => {
        // Limpiar todo el localStorage
        localStorage.clear();
        this.store.dispatch(setUserData({ user: null }));
        this.router.navigate(['/login']);
      })
      .catch((error) => {
        alert('Logout failed. Please try again.');
      });
  }
}
```

Ø=ÜÁ shared\utils

Ø=ÜÄ shared\utils\number-formatter.service.ts

```
import { Injectable } from '@angular/core';

/**
 * Service for formatting numbers, currency, and percentages.
 *
 * This service provides comprehensive number formatting utilities for displaying
 * numeric values in various formats throughout the application. It handles
 * currency formatting, percentage formatting, number formatting with separators,
 * and input value formatting.
```

```
 *
 * Features:
 * - Currency formatting (USD with $ symbol)
 * - Percentage formatting (with % symbol)
 * - Number formatting with thousand separators
 * - Input value formatting (for form inputs)
 * - Currency value parsing (from formatted strings)
 * - Integer formatting
 * - Null/undefined handling
 * - Decimal place control
 *
 * Formatting Methods:
 * - formatCurrency(): Formats as currency with $ symbol
 * - formatPercentage(): Formats as percentage with % symbol
 * - formatNumber(): Formats with thousand separators
 * - formatCurrencyValue(): Formats currency without $ symbol
 * - formatPercentageValue(): Formats percentage without % symbol
 * - formatInteger(): Formats as integer (no decimals)
 * - formatInputValue(): Formats during typing
 * - parseCurrencyValue(): Parses formatted currency back to number
 *
 * Relations:
 * - Used by CurrencyFormatPipe, NumberFormatPipe, PercentageFormatPipe
 * - Used by form components for input formatting
 *
 * @service
 * @injectable
 * @providedIn root
 */
@Injectable({
  providedIn: 'root'
})
export class NumberFormatterService {

  constructor() { }

  /**
   * Formats a number as currency with $ symbol and proper separators
   * @param value - The number to format
   * @param decimals - Number of decimal places (default: 2)
   * @returns Formatted currency string
   */
  formatCurrency(value: number | string | null | undefined): string {
    if (value === null || value === undefined || value === '') {
      return '$0.00';
    }

    const numValue = typeof value === 'string' ? parseFloat(value) : value;

    if (isNaN(numValue)) {
      return '$0.00';
    }

    // Truncate to 2 decimal places
    const truncated = Math.floor(numValue * 100) / 100;

    return new Intl.NumberFormat('en-US', {
      style: 'currency',
      currency: 'USD',
      minimumFractionDigits: 2,
      maximumFractionDigits: 2
    }).format(truncated);
  }

  /**
   * Formats a number as percentage with % symbol
   * @param value - The number to format
   * @param decimals - Number of decimal places (default: 2)
   * @returns Formatted percentage string
   */
```

```
  formatPercentage(value: number | string | null | undefined): string {
    if (value === null || value === undefined || value === '') {
      return '0.00%';
    }

    const numValue = typeof value === 'string' ? parseFloat(value) : value;

    if (isNaN(numValue)) {
      return '0.00%';
    }

    // Truncate to 2 decimal places
    const truncated = Math.floor(numValue * 100) / 100;

    return new Intl.NumberFormat('en-US', {
      style: 'percent',
      minimumFractionDigits: 2,
      maximumFractionDigits: 2
    }).format(truncated / 100);
  }

  /**
   * Formats a number with proper separators and 2 decimal places
   * @param value - The number to format
   * @param decimals - Number of decimal places (default: 2)
   * @returns Formatted number string
   */
  formatNumber(value: number | string | null | undefined, decimals: number = 2): string {
    if (value === null || value === undefined || value === '') {
      return '0.00';
    }

    const numValue = typeof value === 'string' ? parseFloat(value) : value;

    if (isNaN(numValue)) {
      return '0.00';
    }

    // Truncate to specified decimal places
    const truncated = Math.floor(numValue * Math.pow(10, decimals)) / Math.pow(10, decimals);

    return new Intl.NumberFormat('en-US', {
      minimumFractionDigits: decimals,
      maximumFractionDigits: decimals
    }).format(truncated);
  }

  /**
   * Formats a number as currency without the $ symbol (for display purposes)
   * @param value - The number to format
   * @returns Formatted number string with separators
   */
  formatCurrencyValue(value: number | string | null | undefined): string {
    if (value === null || value === undefined || value === '') {
      return '0.00';
    }

    const numValue = typeof value === 'string' ? parseFloat(value) : value;

    if (isNaN(numValue)) {
      return '0.00';
    }

    // Truncate to 2 decimal places
    const truncated = Math.floor(numValue * 100) / 100;

    return new Intl.NumberFormat('en-US', {
      minimumFractionDigits: 2,
      maximumFractionDigits: 2
    }).format(truncated);
```

```
  }

  /**
   * Formats a number as percentage without the % symbol (for display purposes)
   * @param value - The number to format
   * @returns Formatted percentage value string
   */
  formatPercentageValue(value: number | string | null | undefined): string {
    if (value === null || value === undefined || value === '') {
      return '0.00';
    }

    const numValue = typeof value === 'string' ? parseFloat(value) : value;

    if (isNaN(numValue)) {
      return '0.00';
    }

    // Truncate to 2 decimal places
    const truncated = Math.floor(numValue * 100) / 100;

    return new Intl.NumberFormat('en-US', {
      minimumFractionDigits: 2,
      maximumFractionDigits: 2
    }).format(truncated);
  }

  /**
   * Formats a number as an integer (no decimal places) with proper separators
   * @param value - The number to format
   * @returns Formatted integer string
   */
  formatInteger(value: number | string | null | undefined): string {
    if (value === null || value === undefined || value === '') {
      return '0';
    }

    const numValue = typeof value === 'string' ? parseFloat(value) : value;

    if (isNaN(numValue)) {
      return '0';
    }

    // Round to nearest integer
    const rounded = Math.round(numValue);

    return new Intl.NumberFormat('en-US', {
      minimumFractionDigits: 0,
      maximumFractionDigits: 0
    }).format(rounded);
  }

  /**
   * Formats input value during typing (removes non-numeric characters except decimal point)
   * @param input - The input string to clean
   * @returns Cleaned numeric string
   */
  cleanNumericInput(input: string): string {
    return input.replace(/[^0-9.]/g, '');
  }

  /**
   * Formats a number for input display with proper separators during typing
   * @param input - The input string
   * @returns Formatted string with separators
   */
  formatInputValue(input: string): string {
    if (input === '') {
      return '';
    }
```

```
    // Clean the input
    let cleaned = this.cleanNumericInput(input);

    // Avoid more than one decimal point
    const parts = cleaned.split('.');
    if (parts.length > 2) {
      cleaned = parts[0] + '.' + parts[1];
    }

    // Convert to number
    const value = parseFloat(cleaned);
    if (isNaN(value)) {
      return '';
    }

    // Format with separators and preserve decimal places
    return value.toLocaleString('en-US', {
      minimumFractionDigits: parts[1] ? parts[1].length : 0,
      maximumFractionDigits: parts[1] ? parts[1].length : 2
    });
  }

  /**
   * Formats a number as currency for display (on blur)
   * @param value - The number to format
   * @returns Formatted currency string with $ symbol
   */
  formatCurrencyDisplay(value: number | string | null | undefined): string {
    if (value === null || value === undefined || value === '') {
      return '';
    }

    const numValue = typeof value === 'string' ? parseFloat(value) : value;

    if (isNaN(numValue) || numValue <= 0) {
      return '';
    }

    return new Intl.NumberFormat('en-US', {
      style: 'currency',
      currency: 'USD',
      minimumFractionDigits: 2,
      maximumFractionDigits: 2
    }).format(numValue);
  }

  /**
   * Parses a formatted currency string back to a number
   * @param formattedValue - The formatted currency string
   * @returns The numeric value
   */
  parseCurrencyValue(formattedValue: string): number {
    if (!formattedValue) return 0;

    const numericValue = parseFloat(formattedValue.replace(/[^0-9.-]/g, ''));
    return isNaN(numericValue) ? 0 : numericValue;
  }
}
```