**Brandon's Budget - Complete Application Documentation**

This document provides comprehensive documentation for the entire Brandon's Budget iOS application, covering all 75 files with detailed implementation analysis, integration patterns, and debugging guidance.

**Complete Coverage:** All files loaded and documented (75 total files) **Last Updated:** Complete application analysis with full architecture documentation

**📱 App Configuration Files**

**1. App/BrandonsBudget.entitlements**

**File Type:** Property List (Entitlements) **Purpose:** Defines app permissions and capabilities for iOS

**Key Components:**

* **Application Groups:** group.com.brandontitensor.BrandonsBudget
  + Enables data sharing between app and widgets
  + Required for widget functionality
* **Time-Sensitive Notifications:** Enabled
  + Allows budget reminder notifications to bypass Do Not Disturb
  + Critical for notification features

**Integration Points:**

* Referenced by Xcode build system
* Used by NotificationManager for notification permissions
* Required for widget data sharing in Budget\_Widget.swift

**2. App/Brandon-s-Budget-Info.plist**

**File Type:** Property List (App Configuration) **Purpose:** Main app configuration and metadata

**Key Components:**

* **Bundle Configuration:**
  + Display Name: "Brandon's Budget"
  + Bundle ID: Uses $(PRODUCT\_BUNDLE\_IDENTIFIER) variable
  + Version: Uses $(MARKETING\_VERSION) and $(CURRENT\_PROJECT\_VERSION)
* **Device Support:**
  + iOS only (LSRequiresIPhoneOS: true)
  + iPhone: Portrait only
  + iPad: All orientations supported
  + ARM architecture required
* **URL Scheme:** brandonsbudget://
  + Enables deep linking
  + Handled in ContentView.swift via .onOpenURL
* **App Category:** Finance (public.app-category.finance)
* **Notifications:** Permission description for user notifications
* **Encryption:** Non-exempt (ITSAppUsesNonExemptEncryption: false)

**Integration Points:**

* Deep link handling in ContentView.swift
* App metadata accessed in BrandonsBudgetApp.swift extensions
* Notification permissions used by NotificationManager

**3. App/BrandonsBudgetApp.swift**

**File Type:** Swift App Entry Point **Purpose:** Main app entry point with lifecycle management and centralized error handling

**Key State Objects (StateObject):**

swift

@StateObject private var budgetManager = BudgetManager.shared

@StateObject private var themeManager = ThemeManager.shared

@StateObject private var settingsManager = SettingsManager.shared

@StateObject private var errorHandler = ErrorHandler.shared

**Critical Dependencies:**

* BudgetManager.shared - Core data management
* ThemeManager.shared - UI theming
* SettingsManager.shared - App settings
* ErrorHandler.shared - Centralized error handling
* AppStateMonitor.shared - App state tracking
* NotificationManager.shared - Notification system
* CoreDataManager.shared - Data persistence

**Key Lifecycle Methods:**

1. **setupAppOnLaunch()** - Initial app setup
2. **performInitialSetup()** - Async setup tasks:
   * UI appearance configuration
   * Notification setup
   * Data loading
   * Persistence setup
3. **Scene Phase Handling:**
   * Background: Save data, update widgets
   * Inactive: Quick save
   * Active: Refresh data if needed

**Enhanced Background Task Management:**

* handleAppWillTerminate() - Synchronous final save with CoreDataManager.shared.forceSaveSync()
* performBackgroundTasks() - Async coordination of save, widget update, cleanup, and notifications
* handleMemoryWarning() - Error history cleanup and autoreleasepool management
* cleanupTemporaryFiles() - 7-day cache cleanup with FileManager coordination

**Error Handling Integration:**

* Uses AsyncErrorHandler.execute() for async operations
* Global error recovery with data refresh
* Memory warning handling
* Automatic data persistence on app lifecycle events

**Integration Points:**

* Provides environment objects to ContentView
* Coordinates with all manager classes
* Handles system notifications and app state changes

**4. App/ContentView.swift**

**File Type:** SwiftUI Main View **Purpose:** Primary UI container with tab navigation, data persistence, and error recovery

**Environment Dependencies:**

swift

@EnvironmentObject private var budgetManager: BudgetManager

@EnvironmentObject private var themeManager: ThemeManager

@EnvironmentObject private var settingsManager: SettingsManager

@EnvironmentObject private var errorHandler: ErrorHandler

@EnvironmentObject private var appStateMonitor: AppStateMonitor

**State Management:**

* **Tab Navigation:** 4 tabs (Overview, Purchases, History, Settings)
* **Modal Presentations:** Add Purchase, Update Budget, Welcome Popup
* **Loading States:** DataLoadingState enum with idle/loading/loaded/failed
* **Error Recovery:** Global error overlay and recovery mechanisms

**Advanced State Management:**

* DataLoadingState enum with component-specific loading (idle/loading/loaded/failed)
* pendingDeepLink storage for URL handling when app not ready
* lastSaveDate tracking with RelativeDateTimeFormatter display
* restartApp() method for complete state recovery

**Tab Configuration:**

swift

[

("Overview", "chart.pie.fill", AnyView(BudgetOverviewView())),

("Purchases", "cart.fill", AnyView(PurchasesView())),

("History", "clock.fill", AnyView(BudgetHistoryView())),

("Settings", "gear", AnyView(SettingsView()))

]

**Key Features:**

1. **Loading Screen:** Animated loading with progress indication
2. **Floating Action Button:** Context-sensitive quick actions
3. **Deep Link Handling:** URL scheme processing
4. **Scene Phase Management:** Background/foreground transitions
5. **Error Recovery Overlay:** Critical error handling UI

**Data Management Methods:**

* loadInitialData() - Async data loading with error handling
* saveAppData() - Persistent data saving
* refreshAppData() - Data refresh and synchronization
* performGlobalRecovery() - Error recovery workflow

**Deep Link Support:**

* Scheme: brandonsbudget://
* Supported hosts: addPurchase, updateBudget, overview, purchases, history, settings
* Pending link handling when app not ready

**Integration Points:**

* Receives environment objects from BrandonsBudgetApp
* Presents all major feature views
* Coordinates with error handling system
* Manages app-wide navigation state

**5. Brandon.docx - Project Organization**

**File Type:** Documentation **Purpose:** Defines the complete project structure and organization

**Key Organizational Principles:**

* **Feature-Based Architecture:** Each major feature has its own folder with Models/Views/ViewModels
* **Separation of Concerns:** Core logic separate from UI
* **Shared Resources:** Common utilities and constants centralized

**Major Directories:**

1. **📱 App/** - App entry point and configuration
2. **🎯 Core/** - Business logic, data models, services, managers
3. **✨ Features/** - Feature-specific implementations
4. **🎨 Utils/** - Shared utilities and extensions
5. **📱 Widget/** - Widget extension
6. **📦 Resources/** - Assets and localization

**Critical Dependencies Implied:**

* Each feature depends on Core managers
* All views use Utils extensions
* Widget shares data via Core services
* Error handling is centralized across all features

**5a. Package Dependencies (Package.resolved Analysis)**

**SQLite.swift Integration:**

* Version: 0.15.3 (stephencelis/SQLite.swift)
* Revision: a95fc6df17d108bd99210db5e8a9bac90fe984b8
* Used by: CoreDataManager for advanced database operations
* Purpose: High-performance SQLite integration with type safety

**5b. Core Data Model Structure (BudgetModel.xcdatamodel)**

**Entity: BudgetEntryMO**

swift

@objc(BudgetEntryMO)

public class BudgetEntryMO: NSManagedObject {

@NSManaged public var amount: Double

@NSManaged public var category: String?

@NSManaged public var date: Date?

@NSManaged public var id: UUID?

@NSManaged public var note: String?

}

**Entity: MonthlyBudgetMO**

swift

@objc(MonthlyBudgetMO)

public class MonthlyBudgetMO: NSManagedObject {

@NSManaged public var amount: Double

@NSManaged public var category: String?

@NSManaged public var id: UUID?

@NSManaged public var isHistorical: Bool

@NSManaged public var month: Int16

@NSManaged public var year: Int16

}

**Model Configuration:**

* Default values: amount=0.0, month=0, year=0
* Optional fields: category, date, id, note
* Scalar types: Double, Bool, Int16
* Code generation: NSManagedObject subclass
* Syncable: YES for CloudKit compatibility (if enabled)

**5c. Project Configuration (project.pbxproj Analysis)**

**Build Targets:**

* **Main App:** "Brandon's Budget" (com.btlabs.Budget)
* **Widget Extension:** "Budget WidgetExtension" (com.btlabs.Budget.Budget-Widget)

**Key Build Settings:**

* **Deployment:** iOS 17.5+, macOS 14.5+
* **Supported Platforms:** iPhone, iPad (no Mac Catalyst)
* **Code Signing:** Automatic with development team PN22YHBJDB
* **Swift Version:** 5.0
* **Dead Code Stripping:** Enabled for performance

**Framework Dependencies:**

* **Widget Extension:** WidgetKit.framework, SwiftUI.framework, AppIntents.framework
* **File System Sync:** Modern Xcode file organization with synchronized groups

**Critical Build Notes:**

* Widget extension requires separate entitlements file
* App group sharing enabled between main app and widget
* File system synchronization exceptions for proper target membership

**5d. Workspace Configuration Files**

**IDEWorkspaceChecks.plist:**

* IDEDidComputeMac32BitWarning: true - Ensures 64-bit compatibility warnings
* Standard Xcode workspace validation

**WorkspaceSettings.xcsettings:**

* Empty configuration (uses Xcode defaults)
* No custom workspace-level build settings

**🎯 Core Models (Core/Models/)**

**6. Core/Models/BudgetEntry.swift**

**File Type:** SwiftUI Data Model **Purpose:** Represents individual budget transactions with validation and formatting

**Key Components:**

swift

public struct BudgetEntry: Identifiable, Codable, Equatable, Sendable {

public let id: UUID

private let \_amount: Decimal *// Financial precision with thread safety*

private let \_category: String

private let \_date: Date

private let \_note: String?

*// Public computed properties with validation*

public var amount: Double {

(try? \_amount.asDouble()) ?? 0.0

}

*// ... other computed properties*

}

**Validation Rules:**

* Amount must be > 0 and ≤ AppConstants.Validation.maximumTransactionAmount
* Category cannot be empty, max length enforced
* Date cannot be in future
* Note max 500 characters

**Key Features:**

* **Financial Precision:** Uses Decimal internally, exposes Double
* **Thread Safety:** Sendable conformance for concurrent access
* **Validation Errors:** Comprehensive error types with user-friendly messages
* **Formatting Methods:** formattedAmount, formattedDate, shortDate
* **Comparable:** Sorts by date descending (newest first)
* **Testing Support:** Mock generation for development/testing

**Integration Points:**

* Used by BudgetManager for transaction management
* Imported from BudgetEntryMO (Core Data managed object)
* Displayed in transaction lists across multiple views
* Validated against AppConstants.Validation rules

**7. Core/Models/MonthlyBudget.swift**

**File Type:** SwiftUI Data Model **Purpose:** Represents monthly budget allocations per category with comprehensive validation

**Key Components:**

swift

public struct MonthlyBudget: Identifiable, Codable, Equatable {

public let id: UUID

private let \_category: String

private let \_amount: Decimal

private let \_month: Int *// 1-12*

private let \_year: Int *// 1900-9999*

private let \_isHistorical: Bool

}

**Validation System:**

* Category: Non-empty, length validation
* Amount: ≥ 0, max limit enforced
* Month: 1-12 range validation
* Year: 1900-9999 range validation
* Historical validation: Cannot be future date if marked historical

**Date Utilities:**

swift

public var startDate: Date *// First day of month*

public var endDate: Date *// Last day of month*

public var monthYearString: String *// "January 2024"*

**Key Methods:**

* isSameMonth(as:) - Compare with other budgets
* isBefore(*:) / isAfter(*:) - Date comparisons
* Date range calculations for filtering

**Integration Points:**

* Used by BudgetManager for budget management
* Imported from MonthlyBudgetMO (Core Data)
* Powers budget views and planning interfaces
* Sorted chronologically (newest first)

**8. Core/Models/BudgetCategoryData.swift**

**File Type:** SwiftUI Data Model **Purpose:** Calculated budget vs spending data for UI display

**Key Components:**

swift

public struct BudgetCategoryData: Identifiable, Equatable, Hashable {

public let id: UUID

private let \_name: String

private let \_budgeted: Decimal

private let \_spent: Decimal

}

**Calculated Properties:**

swift

public var percentageSpent: Double *// (spent/budgeted) \* 100*

public var isOverBudget: Bool *// spent > budgeted*

public var remaining: Double *// budgeted - spent*

public var percentageRemaining: Double *// (remaining/budgeted) \* 100*

**Use Cases:**

* Budget summary cards
* Progress indicators
* Category comparison views
* Over-budget warnings

**Integration Points:**

* Generated by BudgetManager from raw data
* Used in overview and budget views
* Sorted by budget amount (descending)
* Powers spending analysis features

## Section 9: Core/Models/SpendingData.swift

**File Type:** SwiftUI Data Model  
**Purpose:** Simple chart and visualization data structure for spending analysis

**Key Components:**

swift

public struct SpendingData: Identifiable, Hashable, Codable {

public let id: UUID

public let category: String

public let amount: Double

public let percentage: Double

public let color: Color

public init(category: String, amount: Double, percentage: Double, color: Color) {

self.id = UUID()

self.category = category

self.amount = amount

self.percentage = percentage

self.color = color

}

}

**Validation:**

* Category: Non-empty validation
* Amount: Must be ≥ 0
* Percentage: 0-100 range validation

**Use Cases:**

* Pie charts and visualizations
* Spending breakdowns
* Category analysis
* Historical comparisons

**Integration Points:**

* Generated from spending calculations by BudgetManager
* Used by chart components in Features/
* Sorted by amount (descending)
* Color-coordinated with ThemeManager

**10. Core/Models/BudgetHistoryData.swift**

**File Type:** SwiftUI Data Model **Purpose:** Simple historical budget analysis data structure

**Key Components:**

swift

public struct BudgetHistoryData: Identifiable {

public let id = UUID()

public let category: String

public let budgetedAmount: Double

public let amountSpent: Double

*// Computed analytics*

public var remainingAmount: Double { budgetedAmount - amountSpent }

public var percentageSpent: Double {

guard budgetedAmount > 0 else { return 0 }

return (amountSpent / budgetedAmount) \* 100

}

public var isOverBudget: Bool { amountSpent > budgetedAmount }

}

**Integration Points:**

* Used for historical trend analysis in history views
* Simple data structure for chart and visualization components
* No validation - assumes clean data from BudgetManager

**🏗️ Core Services (Core/Services/)**

## Section 11: Core/Services/CoreDataManager.swift

**File Type:** Core Data Stack Manager  
**Purpose:** Basic Core Data persistence setup with container management

**Architecture:**

swift

@MainActor

public final class CoreDataManager: ObservableObject {

public static let shared = CoreDataManager()

*// Core Data Stack*

private lazy var persistentContainer: NSPersistentContainer = {

let container = NSPersistentContainer(name: "BudgetModel")

container.loadPersistentStores { \_, error in

if let error = error {

fatalError("Core Data failed to load: \(error)")

}

}

return container

}()

public var context: NSManagedObjectContext {

persistentContainer.viewContext

}

}

**Core Data Stack Management:**

* **Model Name:** "BudgetModel" (matches BudgetModel.xcdatamodel)
* **Store Type:** SQLite with default configuration
* **Context:** Main queue context for UI operations
* **Simple Setup:** Basic container initialization

**Basic Operations:**

* Provides managed object context for CRUD operations
* Standard Core Data stack initialization
* Error handling for store loading failures

**Integration Points:**

* Used by BudgetManager for data persistence
* Provides context for all Core Data operations
* Simple shared instance pattern

## Section 12: Core/Services/CSVExport.swift

**File Type:** CSV Export Service Structure  
**Purpose:** Basic export type definitions and configuration

**Architecture:**

swift

public enum CSVExport {

public enum ExportType {

case budgetEntries

case monthlyBudgets

case combined

}

public struct ExportConfiguration {

let timePeriod: TimePeriod

let exportType: ExportType

let includeCurrency: Bool

let dateFormat: String

*// Additional configuration options*

}

}

**Export Types:**

1. **Budget Entries:** Transaction records export
2. **Monthly Budgets:** Budget allocation export
3. **Combined:** Multiple data types in single export

**Configuration Options:**

* Time period filtering via TimePeriod enum
* Currency formatting inclusion
* Date format customization
* Export type selection

**Integration Points:**

* Works with TimePeriod for date filtering
* Used by Settings export functionality
* Provides structure for export operations

## Section 13: Core/Services/CSVImport.swift

**File Type:** CSV Import Service Structure  
**Purpose:** Basic import data structures and configuration

**Architecture:**

swift

public enum CSVImport {

public enum ImportType {

case budgetEntries

case monthlyBudgets

case autoDetect

}

public struct ImportConfiguration {

let importType: ImportType

let validateDuplicates: Bool

let skipInvalidRows: Bool

let maxFileSize: Int64

*// Additional import options*

}

}

**Import Data Structures:**

swift

public struct PurchaseImportData: Codable, Equatable {

public let date: String

public let amount: Double

public let category: String

public let note: String?

func toBudgetEntry(dateFormatter: DateFormatter) throws -> BudgetEntry {

*// Conversion logic to BudgetEntry*

}

}

public struct BudgetImportData: Codable, Equatable {

public let year: Int

public let month: Int

public let category: String

public let amount: Double

public let isHistorical: Bool

func toMonthlyBudget() throws -> MonthlyBudget {

*// Conversion logic to MonthlyBudget*

}

}

**Import Configuration:**

* Import type selection (budgets, purchases, auto-detect)
* Duplicate validation options
* Error handling preferences (skip vs strict)
* File size limitations

**Integration Points:**

* Used by Settings import functionality
* Creates BudgetEntry and MonthlyBudget objects
* Coordinates with BudgetManager for data persistence

**14. Core/Services/ErrorHandlingSystem.swift**

**File Type:** Centralized Error Management **Purpose:** Comprehensive error handling with user-friendly messages and recovery

**Architecture:**

swift

@MainActor

public final class ErrorHandler: ObservableObject {

public static let shared = ErrorHandler()

@Published public var currentError: AppError?

@Published public var isShowingError = false

@Published public var errorHistory: [ErrorEntry] = []

private let maxHistoryCount = 50

public struct ErrorEntry: Identifiable {

public let id = UUID()

public let error: AppError

public let timestamp: Date

public let context: String?

}

}

**Enhanced Error Types:**

swift

public enum AppError: LocalizedError, Identifiable {

case dataLoad(underlying: Error)

case dataSave(underlying: Error)

case csvImport(underlying: Error)

case csvExport(underlying: Error)

case validation(message: String)

case network(underlying: Error)

case permission(type: PermissionType)

case fileAccess(underlying: Error)

case generic(message: String)

public enum PermissionType {

case notifications, fileAccess, camera, photos

}

*// Smart error transformation*

public static func from(\_ error: Error) -> AppError {

*// Maps NSError domains to AppError cases*

*// Handles BudgetManager.BudgetManagerError*

*// Processes CSVImport/Export errors*

*// Falls back to generic for unknown errors*

}

}

**Error Severity System:**

swift

public enum ErrorSeverity {

case info, warning, error, critical

var color: Color *// UI color coding*

var icon: String *// System icon names*

}

**UI Component System:**

* **ErrorAlert:** ViewModifier for standard alert presentation with retry
* **InlineErrorView:** Rich inline error with retry/dismiss buttons
* **ErrorToast:** Animated toast with auto-dismiss and haptic feedback
* **ErrorHistoryView:** Complete error history browser with filtering

**Advanced Features:**

* **Haptic Feedback:** Heavy impact for critical errors
* **Error History:** 50-item rolling history with timestamps
* **Context Tracking:** Optional context strings for debugging
* **Retry Logic:** Configurable retry options for recoverable errors
* **AsyncErrorHandler:** Wrapper for async operations with automatic error handling

**Integration Points:**

* Used throughout the app via ErrorHandler.shared
* Integrated with all manager classes
* Provides UI modifiers for error display
* Coordinates with notification system for error alerts

**👥 Core Managers (Core/Managers/)**

## Section 15: Core/Managers/BudgetManager.swift

**File Type:** Core Business Logic Manager  
**Purpose:** Central coordinator for budget-related operations with basic state management

**Architecture:**

swift

@MainActor

public final class BudgetManager: ObservableObject {

public static let shared = BudgetManager()

@Published public private(set) var entries: [BudgetEntry] = []

@Published public private(set) var monthlyBudgets: [MonthlyBudget] = []

@Published public private(set) var isLoading = false

private let coreDataManager = CoreDataManager.shared

}

**Core Data Operations:**

swift

*// Entry Management*

func getEntries(for period: TimePeriod? = nil,

category: String? = nil,

sortedBy: BudgetSortOption = .date,

ascending: Bool = false) async throws -> [BudgetEntry]

func addEntry(\_ entry: BudgetEntry) async throws

func updateEntry(\_ entry: BudgetEntry) async throws

func deleteEntry(\_ entry: BudgetEntry) async throws

*// Budget Management*

func getMonthlyBudgets(for month: Int, year: Int) -> [MonthlyBudget]

func addOrUpdateMonthlyBudget(\_ budget: MonthlyBudget) async throws

func updateMonthlyBudgets(\_ budgets: [MonthlyBudget], for month: Int, year: Int) async throws

**Category Management:**

* getAvailableCategories() -> [String]
* Basic category retrieval from existing data
* Category validation and management

**State Management:**

* Published properties for reactive UI updates
* Loading state tracking
* Error handling through throws

**Integration Points:**

* Primary interface for all budget data operations
* Coordinates with CoreDataManager for persistence
* Provides data to all UI views via @EnvironmentObject
* Used by import/export services for data processing

**Error Management:**

* Uses standard Swift error throwing
* Integrates with app-wide error handling
* Provides meaningful error context for UI display

**16. Core/Managers/SettingsManager.swift**

**File Type:** App Settings & Preferences Manager **Purpose:** Comprehensive settings management with validation, backup, and migration

**Architecture:**

swift

@MainActor

public final class SettingsManager: ObservableObject {

public static let shared = SettingsManager()

*// Core Settings*

@Published public var userName: String

@Published public var defaultCurrency: String

@Published public var notificationsAllowed: Bool

*// Advanced Settings*

@Published public var enableDataBackup: Bool

@Published public var privacyMode: Bool

@Published public var biometricAuthEnabled: Bool

}

**Settings Categories:**

1. **User Preferences:**
   * User name (1-50 chars, validation)
   * Default currency (supported: USD, EUR, GBP, JPY, etc.)
   * Privacy mode toggle
   * Haptic feedback preferences
2. **Notification Settings:**

swift

public enum PurchaseNotificationFrequency: String, CaseIterable {

case daily, weekly, monthly

}

public enum BudgetTotalNotificationFrequency: String, CaseIterable {

case monthly, yearly

}

1. **Data Management:**

swift

public enum BackupFrequency: String, CaseIterable {

case never, daily, weekly, monthly

var intervalInSeconds: TimeInterval?

}

public enum DataExportFormat: String, CaseIterable {

case csv, json

}

1. **Display Options:**
   * Show decimal places toggle
   * Round to nearest cent
   * Advanced features toggle
   * Biometric authentication

**Advanced Features:**

1. **Settings Migration System:**

swift

private let currentSettingsVersion = 3

private func performMigrationIfNeeded() {

let savedVersion = userDefaults.integer(forKey: Keys.settingsVersion.key)

if savedVersion < currentSettingsVersion {

try migrateSettings(from: savedVersion, to: currentSettingsVersion)

*// Version 1: Haptic feedback*

*// Version 2: Privacy mode and backup*

*// Version 3: Biometric auth and advanced features*

}

}

1. **Comprehensive Backup System:**

swift

func createBackup() async throws -> URL {

*// JSON export with device metadata*

*// 30-day retention cleanup*

*// Settings validation and integrity checks*

*// File system coordination with error recovery*

}

func restoreFromBackup(\_ url: URL) async throws {

*// Version compatibility checking*

*// Incremental setting restoration*

*// Validation before final application*

}

1. **Validation Rules:**

swift

private struct ValidationRules {

static let maxUserNameLength = 50

static let supportedCurrencies = ["USD", "EUR", "GBP", "JPY", "CAD", "AUD", "CHF", "CNY", "SEK", "NOK", "DKK"]

static let reservedUserNames = ["admin", "system", "user", "test", "guest"]

}

1. **Performance Monitoring:**

* Operation timing metrics for all settings operations
* Background queue processing (settingsQueue, metricsQueue)
* Automatic backup scheduling with timer management
* Memory optimization with cleanup

**Key Methods:**

swift

func updateNotificationSettings(allowed:purchaseEnabled:purchaseFrequency:budgetEnabled:budgetFrequency:) async throws

func resetToDefaults() async throws

func exportSettings() async throws -> [String: Any]

func importSettings(\_:) async throws

func createBackup() async throws -> URL

func restoreFromBackup(\_:) async throws

**Integration Points:**

* Coordinates with NotificationManager for notification scheduling
* Provides currency formatting preferences
* Manages theme and display preferences
* Integrates with backup and data export systems

**17. Core/Managers/NotificationManager.swift**

**File Type:** Advanced Notification System Manager **Purpose:** Comprehensive notification management with system diagnostics and smart scheduling

**Architecture:**

swift

@MainActor

public final class NotificationManager: NSObject, ObservableObject, UNUserNotificationCenterDelegate {

public static let shared = NotificationManager()

@Published public private(set) var authorizationStatus: UNAuthorizationStatus = .notDetermined

@Published public private(set) var pendingNotifications: [UNNotificationRequest] = []

@Published public private(set) var deliveredNotifications: [UNNotification] = []

@Published public private(set) var lastError: AppError?

@Published public private(set) var isProcessing = false

}

**Advanced Type System:**

swift

public enum NotificationCategory: String, CaseIterable {

case purchase, budget, reminder, achievement, warning

var displayName: String { rawValue.capitalized }

var systemImageName: String { */\* icon mapping \*/* }

}

public enum NotificationPriority: String, CaseIterable {

case low, normal, high, critical

var interruptionLevel: UNNotificationInterruptionLevel { */\* iOS 15+ support \*/* }

var sound: UNNotificationSound? { */\* priority-based sounds \*/* }

}

public struct NotificationTemplate {

let title: String, body: String, category: NotificationCategory

let priority: NotificationPriority, actions: [UNNotificationAction]

let userInfo: [String: Any]

func validate() throws { */\* comprehensive validation \*/* }

}

**Smart Scheduling System:**

* Purchase reminders with frequency-based triggers
* Budget warnings at 80% and 100% thresholds
* Achievement notifications with 2-second delays
* Smart budget notifications based on spending patterns
* System health monitoring with comprehensive diagnostics

**Performance Features:**

* 64-notification iOS limit management
* 7-day old notification cleanup
* Performance metrics with timing
* Background processing with error recovery
* Comprehensive system diagnostic capabilities

**Integration Points:**

* UNUserNotificationCenterDelegate for action handling
* Deep link coordination via NotificationCenter.default
* SettingsManager integration for preference management
* ErrorHandler integration for centralized error management

**18. Core/Managers/ThemeManager.swift**

**File Type:** UI Theme & Color Management **Purpose:** Centralized theme management with color coordination and system integration

**Architecture:**

swift

@MainActor

public final class ThemeManager: ObservableObject {

public static let shared = ThemeManager()

@Published public var primaryColor: Color

@Published public var isDarkMode: Bool

}

**Color System:**

1. **Primary Colors:**

swift

public struct ColorOption: Identifiable, Hashable, Codable {

public let id: UUID

let name: String

let colorComponents: ColorComponents

var color: Color

}

public static let availableColors: [ColorOption] = [

ColorOption(name: "Blue", color: .blue),

ColorOption(name: "Purple", color: .purple),

ColorOption(name: "Green", color: .green),

*// ... 8 total colors*

]

1. **Category Colors:**
   * Deterministic color assignment based on category name hash
   * 7 distinct colors for category differentiation
   * Consistent across app sessions
2. **Semantic Colors:**

swift

public struct SemanticColors {

var success: Color { .green }

var warning: Color { .orange }

var error: Color { .red }

var info: Color { primaryColor }

var backgroundPrimary: Color { Color(.systemBackground) }

var textPrimary: Color { Color(.label) }

*// ... system-aware colors*

}

**System Integration:**

swift

private func updateGlobalAppearance() {

*// Navigation bar appearance with primary color coordination*

let appearance = UINavigationBarAppearance()

appearance.configureWithDefaultBackground()

appearance.largeTitleTextAttributes = [

.foregroundColor: UIColor(primaryColor)

]

appearance.titleTextAttributes = [

.foregroundColor: UIColor(primaryColor)

]

*// Apply to all navigation bar types*

UINavigationBar.appearance().standardAppearance = appearance

UINavigationBar.appearance().scrollEdgeAppearance = appearance

UINavigationBar.appearance().compactAppearance = appearance

*// Tab bar appearance coordination*

let tabBarAppearance = UITabBarAppearance()

tabBarAppearance.configureWithDefaultBackground()

UITabBar.appearance().standardAppearance = tabBarAppearance

UITabBar.appearance().scrollEdgeAppearance = tabBarAppearance

}

**Storage System:**

* UserDefaults with StorageKeys.primaryColorName for color persistence
* Color name mapping to availableColors array for type safety
* Automatic theme application on initialization

**Key Methods:**

swift

func resetToDefaults()

func colorForCategory(\_ category: String) -> Color

func primaryGradient() -> LinearGradient

func updateTheme(with colorOption: ColorOption)

**Integration Points:**

* Used by all UI components for consistent theming
* Coordinates with SettingsManager for user preferences
* Provides category color coordination
* Updates system appearance elements

**🔧 Core Types (Core/Types/)**

**19. Core/Types/AppEnums.swift**

**File Type:** Shared Enumerations **Purpose:** Centralized enums for consistent app-wide behavior

**Major Enumerations:**

1. **Sorting & Filtering:**

swift

public enum BudgetSortOption: String, CaseIterable, Codable {

case category = "Category"

case budgetedAmount = "Budgeted Amount"

case amountSpent = "Amount Spent"

case date = "Date"

case amount = "Amount"

}

public enum FilterType: String, CaseIterable, Codable {

case all = "All"

case category = "Category"

case date = "Date"

case amount = "Amount"

}

public enum SortDirection: String, CaseIterable, Codable {

case ascending = "Ascending"

case descending = "Descending"

var isAscending: Bool

}

1. **Data Management:**

swift

public enum ExportFormat: String, CaseIterable, Codable {

case csv = "CSV"

case json = "JSON"

case pdf = "PDF"

var fileExtension: String

var mimeType: String

}

public enum ImportFormat: String, CaseIterable, Codable {

case csv = "CSV"

case json = "JSON"

var allowedContentTypes: [String]

}

1. **UI Configuration:**

swift

public enum ViewType: String, CaseIterable, Codable {

case list = "List"

case chart = "Chart"

case summary = "Summary"

var systemImageName: String

}

public enum ChartType: String, CaseIterable, Codable {

case pie = "Pie"

case bar = "Bar"

case line = "Line"

case donut = "Donut"

}

**Integration Points:**

* Used throughout the app for consistent behavior
* Provides display names and system icons
* Supports coding/decoding for persistence
* Type-safe alternatives to string constants

**20. Core/Types/TimePeriod.swift**

**File Type:** Time Period Management **Purpose:** Comprehensive date range handling with calendar integration

**Architecture:**

swift

public enum TimePeriod: Equatable, Hashable, Codable, Sendable {

case today, yesterday, thisWeek, lastWeek

case thisMonth, lastMonth, thisQuarter, lastQuarter

case thisYear, lastYear

case last7Days, last30Days, last90Days, last12Months

case allTime

case custom(Date, Date)

}

**Key Features:**

1. **Date Interval Calculation:**

swift

public func dateInterval() -> DateInterval

* Handles all calendar calculations
* Timezone-aware date math
* Quarter and week boundary logic

1. **Period Properties:**

swift

public var durationInDays: Int

public var includesCurrentDate: Bool

public var isRelative: Bool *// Changes with current date*

public var isComplete: Bool *// Past vs current periods*

1. **Navigation:**

swift

public func next() -> TimePeriod? *// Next period of same type*

public func previous() -> TimePeriod? *// Previous period of same type*

1. **Formatting:**

swift

public var displayName: String *// "This Month"*

public var shortDisplayName: String *// "This Month" → "Month"*

public var systemImageName: String *// Calendar icons*

func formattedDateRange(style:) -> String

**Static Collections:**

swift

static var allCases: [TimePeriod] *// All available periods*

static var commonPeriods: [TimePeriod] *// Frequently used*

static var recentPeriods: [TimePeriod] *// Today, yesterday, etc.*

static var historicalPeriods: [TimePeriod] *// Past periods*

**Integration Points:**

* Used by filtering systems throughout the app
* Powers date range pickers in UI
* Used by export/import for time-based filtering
* Supports data analysis and reporting features

**21. Core/Types/SharedDataManager.swift**

**File Type:** Widget Data Sharing Manager **Purpose:** App-Widget data synchronization with comprehensive validation

**Architecture:**

swift

@MainActor

public final class SharedDataManager: ObservableObject {

public static let shared = SharedDataManager()

@Published public private(set) var currentBudgetSummary: BudgetSummary?

@Published public private(set) var recentTransactions: [RecentTransaction] = []

@Published public private(set) var topCategories: [CategorySpending] = []

}

**Data Structures:**

1. **Budget Summary:**

swift

public struct BudgetSummary: Codable, Equatable {

public let monthlyBudget: Double

public let totalSpent: Double

public let remainingBudget: Double

public let percentageUsed: Double

public let isOverBudget: Bool

public let categoryCount: Int

public let transactionCount: Int

public let lastUpdated: Date

public let currentMonth: String

}

1. **Recent Transactions:**

swift

public struct RecentTransaction: Codable, Identifiable, Equatable {

public let id: UUID

public let amount: Double

public let category: String

public let date: Date

public let note: String?

}

1. **Category Spending:**

swift

public struct CategorySpending: Codable, Identifiable, Equatable {

public let name: String

public let amount: Double

public let percentage: Double

public let color: String

}

**Widget Integration:**

1. **Complete Widget Data:**

swift

public struct WidgetData: Codable, Equatable {

public let budgetSummary: BudgetSummary

public let recentTransactions: [RecentTransaction] *// Max 5*

public let topCategories: [CategorySpending] *// Max 5*

public let lastUpdated: Date

public let appVersion: String

}

1. **App Group Storage:**
   * Uses group.com.brandontitensor.BrandonsBudget
   * JSON encoding for data persistence
   * UserDefaults-based shared storage
   * Data validation on read/write
2. **Health Monitoring:**

swift

public struct DataHealth {

public enum Status {

case healthy, warning, error, critical

}

public let status: Status

public let message: String

public let recommendation: String?

}

**Key Methods:**

swift

func updateBudgetData(monthlyBudget:totalSpent:remainingBudget:categoryCount:transactionCount:) async throws

func updateRecentTransactions(\_:) async throws

func updateTopCategories(\_:) async throws

func updateCompleteWidgetData(budgetSummary:transactions:categories:) async throws

**Error Handling:**

swift

public enum SharedDataError: LocalizedError {

case invalidData(String), encodingFailed, decodingFailed

case userDefaultsUnavailable, dataCorrupted, widgetUpdateFailed

}

**Integration Points:**

* Updated by BudgetManager when data changes
* Triggers WidgetCenter.shared.reloadAllTimelines()
* Provides data health monitoring
* Supports backup/restore operations
* Used by widget extension for data display

**✨ Budget Feature (Features/Budget/)**

# Section 22: Features/Budget/ViewModels/BudgetViewModel.swift

**File Type:** Advanced Budget Management Logic  
**Purpose:** Comprehensive budget state management with real-time validation, analytics, performance monitoring, and sophisticated error handling

## Architecture Overview:

swift

@MainActor

public final class BudgetViewModel: ObservableObject {

*// Core Dependencies*

private let budgetManager: BudgetManager

private let errorHandler: ErrorHandler

private let settingsManager: SettingsManager

*// Published State*

@Published public private(set) var viewState: ViewState = .idle

@Published public private(set) var monthlyBudgets: [Int: [String: Double]] = [:]

@Published public private(set) var budgetSummary: BudgetSummary?

@Published public private(set) var budgetAnalytics: BudgetAnalytics?

@Published public private(set) var hasUnsavedChanges = false

@Published public private(set) var validationErrors: [String: [String]] = [:]

@Published public private(set) var isProcessing = false

*// Input State*

@Published public var selectedYear: Int

@Published public var selectedMonth: Int

@Published public var newCategoryName = ""

@Published public var newCategoryAmount: Double = 0

}

## Advanced State Management:

### View State System:

swift

public enum ViewState: Equatable {

case idle

case loading

case loaded

case empty

case error(AppError)

var isLoading: Bool {

if case .loading = self { return true }

return false

}

var hasError: Bool {

if case .error = self { return true }

return false

}

}

public enum OperationType {

case loadBudgets, saveBudgets, addCategory, updateCategory, deleteCategory

case validateCategory, calculateAnalytics

}

## Comprehensive Analytics System:

### Budget Summary:

swift

public struct BudgetSummary: Equatable {

public let totalYearlyBudget: Double

public let totalMonthlyBudget: Double

public let categoryCount: Int

public let averageMonthlyBudget: Double

public let largestCategory: String?

public let smallestCategory: String?

*// Computed properties*

public var isValid: Bool { categoryCount > 0 && totalMonthlyBudget > 0 }

public var formattedYearlyBudget: String { totalYearlyBudget.asCurrency }

public var formattedMonthlyBudget: String { totalMonthlyBudget.asCurrency }

public var averageCategoryBudget: Double {

categoryCount > 0 ? totalMonthlyBudget / Double(categoryCount) : 0

}

}

### Advanced Analytics:

swift

public struct BudgetAnalytics: Equatable {

public let totalBudget: Double

public let averageCategoryBudget: Double

public let categoryCount: Int

public let monthlyDistribution: [Int: Double]

public let categoryDistribution: [String: Double]

public let largestCategory: (key: String, value: Double)?

public let smallestCategory: (key: String, value: Double)?

*// Advanced metrics*

public var isBalanced: Bool { monthlyVariation < 20.0 }

public var monthlyVariation: Double {

*// Calculate coefficient of variation across months*

let values = Array(monthlyDistribution.values)

guard values.count > 1 else { return 0 }

let mean = values.reduce(0, +) / Double(values.count)

let variance = values.map { pow($0 - mean, 2) }.reduce(0, +) / Double(values.count)

return sqrt(variance) / mean \* 100

}

public var efficiencyScore: Double {

*// Score based on budget distribution and variance*

let balanceScore = isBalanced ? 1.0 : 0.5

let utilizationScore = totalBudget > 0 ? min(1.0, totalBudget / 10000) : 0

return (balanceScore + utilizationScore) / 2.0

}

}

## Real-Time Validation System:

### Validation with Debouncing:

swift

private var cancellables = Set<AnyCancellable>()

private func setupValidation() {

*// Debounced validation for new category input*

$newCategoryName.combineLatest($newCategoryAmount)

.debounce(for: .milliseconds(300), scheduler: DispatchQueue.main)

.sink { [weak self] \_, \_ in

self?.validateNewCategoryInput()

}

.store(in: &cancellables)

*// Real-time validation for existing categories*

$monthlyBudgets

.debounce(for: .milliseconds(500), scheduler: DispatchQueue.main)

.sink { [weak self] \_ in

self?.validateAllCategories()

}

.store(in: &cancellables)

}

### Category Validation:

swift

public struct CategoryValidation: Equatable {

public let isValid: Bool

public let errors: [String]

public let warnings: [String]

public var hasErrors: Bool { !errors.isEmpty }

public var hasWarnings: Bool { !warnings.isEmpty }

public var hasIssues: Bool { hasErrors || hasWarnings }

}

public func validateCategory(\_ categoryName: String, amount: Double) -> CategoryValidation {

var errors: [String] = []

var warnings: [String] = []

*// Name validation*

let trimmedName = categoryName.trimmingCharacters(in: .whitespacesAndNewlines)

if trimmedName.isEmpty {

errors.append("Category name cannot be empty")

} else if trimmedName.count < 2 {

errors.append("Category name must be at least 2 characters")

} else if trimmedName.count > AppConstants.Validation.maxCategoryNameLength {

errors.append("Category name is too long")

}

*// Duplicate check*

if getAllCategoryNames().contains(trimmedName) {

errors.append("Category name already exists")

}

*// Amount validation*

if amount < 0 {

errors.append("Budget amount cannot be negative")

} else if amount == 0 {

warnings.append("Budget amount is zero - category will not appear in reports")

} else if amount > AppConstants.Validation.maximumTransactionAmount {

errors.append("Budget amount exceeds maximum limit")

}

*// Reasonableness checks*

if amount > 10000 {

warnings.append("Large budget amount - please verify")

}

return CategoryValidation(isValid: errors.isEmpty, errors: errors, warnings: warnings)

}

## Performance Monitoring:

### Operation Metrics:

swift

private var operationMetrics: [String: TimeInterval] = [:]

private let metricsQueue = DispatchQueue(label: "com.brandonsbudget.budgetvm.metrics", qos: .utility)

private func recordMetric(\_ operation: String, duration: TimeInterval) {

metricsQueue.async {

self.operationMetrics[operation] = duration

#if DEBUG

if duration > 1.0 {

print("⚠️ BudgetViewModel: Slow operation '\(operation)' took \(String(format: "%.2f", duration \* 1000))ms")

}

#endif

}

}

private func performanceOptimizedOperation<T>(\_ operationType: OperationType, operation: () async throws -> T) async throws -> T {

let startTime = Date()

isProcessing = true

defer {

isProcessing = false

recordMetric(String(describing: operationType), duration: Date().timeIntervalSince(startTime))

}

return try await operation()

}

## Background Processing with AsyncErrorHandler:

### Async Operations:

swift

public func loadBudgets() async {

await performanceOptimizedOperation(.loadBudgets) {

viewState = .loading

clearValidationErrors()

do {

let budgets = try await budgetManager.getMonthlyBudgets(for: selectedMonth, year: selectedYear)

await MainActor.run {

self.monthlyBudgets = self.organizeBudgetsByMonth(budgets)

self.calculateBudgetSummary()

self.calculateBudgetAnalytics()

self.viewState = budgets.isEmpty ? .empty : .loaded

self.hasUnsavedChanges = false

}

} catch {

await handleError(AppError.from(error), context: "Loading budgets")

}

}

}

public func addCategory(includeFutureMonths: Bool = false) async {

let validation = validateCategory(newCategoryName, amount: newCategoryAmount)

guard validation.isValid else {

await MainActor.run {

validationErrors["newCategory"] = validation.errors

}

return

}

await performanceOptimizedOperation(.addCategory) {

do {

try await budgetManager.addCategory(

newCategoryName.trimmingCharacters(in: .whitespacesAndNewlines),

amount: newCategoryAmount,

month: selectedMonth,

year: selectedYear,

includeFutureMonths: includeFutureMonths

)

await MainActor.run {

self.newCategoryName = ""

self.newCategoryAmount = 0

self.hasUnsavedChanges = true

}

await loadBudgets() *// Refresh data*

} catch {

await handleError(AppError.from(error), context: "Adding category")

}

}

}

## Advanced Category Management:

### Multi-Month Operations:

swift

public func updateCategory(

oldName: String,

newName: String,

newAmount: Double,

applyToFutureMonths: Bool = false

) async {

let validation = validateCategory(newName, amount: newAmount)

guard validation.isValid else {

await MainActor.run {

validationErrors[oldName] = validation.errors

}

return

}

await performanceOptimizedOperation(.updateCategory) {

do {

if applyToFutureMonths {

*// Update current and all future months*

let futureMonths = generateFutureMonths(from: selectedMonth, year: selectedYear)

for (month, year) in futureMonths {

try await budgetManager.updateMonthlyBudget(

category: oldName,

newCategory: newName,

amount: newAmount,

month: month,

year: year

)

}

} else {

*// Update only current month*

try await budgetManager.updateMonthlyBudget(

category: oldName,

newCategory: newName,

amount: newAmount,

month: selectedMonth,

year: selectedYear

)

}

await loadBudgets() *// Refresh data*

} catch {

await handleError(AppError.from(error), context: "Updating category")

}

}

}

## Memory Management and Cache Invalidation:

### Cache Management:

swift

private var analyticsCache: (budgets: [Int: [String: Double]], analytics: BudgetAnalytics)?

private var summaryCache: (budgets: [Int: [String: Double]], summary: BudgetSummary)?

private func invalidateCache() {

analyticsCache = nil

summaryCache = nil

}

private func calculateBudgetAnalytics() {

*// Use cache if budgets haven't changed*

if let cache = analyticsCache, cache.budgets == monthlyBudgets {

budgetAnalytics = cache.analytics

return

}

let startTime = Date()

*// Perform expensive calculations*

let analytics = generateAnalytics(from: monthlyBudgets)

*// Cache the result*

analyticsCache = (budgets: monthlyBudgets, analytics: analytics)

budgetAnalytics = analytics

recordMetric("calculateBudgetAnalytics", duration: Date().timeIntervalSince(startTime))

}

## Error Handling and Recovery:

### Comprehensive Error Management:

swift

private func handleError(\_ error: AppError, context: String) async {

await MainActor.run {

viewState = .error(error)

errorHandler.handle(error, context: "BudgetViewModel: \(context)")

*// Clear processing state*

isProcessing = false

*// Provide recovery suggestions based on error type*

switch error {

case .dataSave:

*// Suggest retry with auto-save disabled*

break

case .validation(let message):

*// Add to validation errors for specific field display*

validationErrors["general"] = [message]

default:

break

}

}

}

public func retryLastOperation() async {

clearValidationErrors()

await loadBudgets()

}

## Integration Points:

### Manager Coordination:

* **BudgetManager:** Primary data source with real-time coordination
* **ErrorHandler:** Centralized error management with context
* **SettingsManager:** User preferences and validation rules

### UI Coordination:

* **Published Properties:** Reactive UI updates via Combine
* **Loading States:** Non-blocking operation feedback
* **Validation Display:** Real-time validation with field-specific errors

### Performance Integration:

* **Background Processing:** Heavy operations off main thread
* **Debounced Updates:** Prevents excessive validation cycles
* **Cache Management:** Automatic invalidation and regeneration

## Testing Support:

swift

#if DEBUG

extension BudgetViewModel {

func loadTestData() async {

*// Load realistic test budget data*

await loadBudgets()

}

func simulateSlowOperation() async {

await Task.sleep(nanoseconds: 2\_000\_000\_000) *// 2 seconds*

}

func getValidationStateForTesting() -> [String: [String]] {

return validationErrors

}

func getPerformanceMetricsForTesting() -> [String: TimeInterval] {

return metricsQueue.sync { operationMetrics }

}

}

#endif

This BudgetViewModel represents **enterprise-level state management** with sophisticated validation, performance monitoring, and error handling patterns that serve as a model for other ViewModels throughout the application.

**23. Features/Budget/Views/BudgetView.swift**

**File Type:** Budget Management Interface **Purpose:** Comprehensive budget editing with year/month navigation and category management

**Architecture:**

swift

struct BudgetView: View {

@EnvironmentObject private var budgetManager: BudgetManager

@EnvironmentObject private var themeManager: ThemeManager

@EnvironmentObject private var errorHandler: ErrorHandler

@State private var selectedYear: Int

@State private var selectedMonth: Int

@State private var monthlyBudgets: [Int: [String: Double]] = [:]

}

**Key UI Components:**

1. **Year Section:** Year picker with total yearly budget display
2. **Month Tabs:** Horizontal scrollable month selector with budget totals
3. **Summary Section:** Monthly budget and category count cards
4. **Categories Section:** List of budget categories with navigation to edit
5. **Add Category Button:** Prominent action button for new categories

**Data Loading States:**

swift

private enum DataLoadingState {

case idle, loading, loaded, failed(AppError)

}

**Category Management:**

* **Add Category Sheet:** Modal with name/amount input and future month options
* **Edit Integration:** Navigation to EditCategoryView for modifications
* **Future Month Handling:** Confirmation dialogs for applying changes to future periods
* **Validation:** Real-time input validation with error display

**Performance Features:**

* **Auto-save:** Changes tracked and saved automatically
* **Loading Overlays:** Non-blocking processing indicators
* **Pull-to-Refresh:** Manual data refresh capability
* **Error Recovery:** Section-specific error handling with retry

**Integration Points:**

* Uses BudgetManager for all data operations
* Coordinates with MoneyCalculatorView for amount input
* Integrates with EditCategoryView for category modifications
* Handles future month confirmation workflows

**24. Features/Budget/Views/EditCategoryView.swift**

**File Type:** Category Editing Interface **Purpose:** Detailed category editing with validation, deletion, and future month coordination

**Architecture:**

swift

struct EditCategoryView: View {

let monthlyBudgets: [Int: [String: Double]]

let initialCategory: String

let month: Int, year: Int

let onUpdate: (String, String, Double) -> Void

@State private var categoryName: String

@State private var amount: Double

@State private var hasUnsavedChanges = false

}

**Validation System:**

* **Real-time Validation:** Name and amount validation on change
* **Duplicate Detection:** Prevents category name conflicts
* **Required Category Protection:** Cannot delete system-required categories
* **Amount Limits:** Enforces maximum transaction amount limits

**Key Features:**

1. **Category Details Section:**
   * Editable name field with character limit
   * Amount display with calculator integration
   * Real-time validation messages
2. **Information Section:**
   * Current month/year display
   * Save status indicator
   * Required category protection notice
3. **Delete Section:**
   * Conditional delete button (non-required categories only)
   * Confirmation workflow with warning about data migration
   * Future month deletion options

**Confirmation Workflows:**

* **Discard Changes:** Alert when attempting to cancel with unsaved changes
* **Delete Category:** Multi-step confirmation with impact explanation
* **Future Months:** Options for applying deletions to future periods

**Integration Points:**

* Coordinates with MoneyCalculatorView for amount editing
* Uses BudgetManager for category operations
* Integrates with parent view through callback pattern
* Follows established error handling patterns

# **25. Features/Budget/Views/Components/BudgetCategoryRow.swift**

**File Type:** Budget Category Display Component  
**Purpose:** Reusable row component for displaying budget categories with progress visualization and interaction support

**Architecture Overview:**

swift

struct BudgetCategoryRow: View {

let category: String

let amount: Double

@EnvironmentObject private var themeManager: ThemeManager

*// Visual components*

private var categoryLabel: some View

private var amountLabel: some View

private var progressBar: some View *// Future integration for spent amount display*

}

**Key Integration Points:**

**Data Dependencies:**

* **Input:** Category name (String) and budget amount (Double)
* **Environment:** ThemeManager for consistent color coordination
* **Future:** Designed for BudgetHistoryData integration (spent amounts)

**Visual Design System:**

* **Color Indicator:** Small circle using themeManager.colorForCategory(category)
* **Typography:** Headline font for category name, subheadline for amount
* **Layout:** HStack with color indicator, category name, and amount display

**Theme Integration:**

swift

*// Color coordination with theme system*

Circle()

.fill(themeManager.colorForCategory(category))

.frame(width: 12, height: 12)

**Current Implementation Status:**

* **Basic Layout:** Category name and amount display ✅
* **Color Coordination:** Theme-based category colors ✅
* **Progress Visualization:** TODO comment indicates future spent amount integration
* **Accessibility:** Combined element with descriptive labels ✅

**Integration Dependencies:**

* **ThemeManager:** Required for colorForCategory() method
* **Amount Formatting:** Uses Double.asCurrency extension
* **Future Integration:** Designed to work with BudgetHistoryData for progress bars

**Usage Patterns:**

* Used in budget views for consistent category display
* Supports navigation to category detail views
* Part of list-based budget category management interfaces

**Section 26: Features/Budget/Views/Components/BudgetSummaryCard.swift**

**File Type:** Budget Summary Display Component  
**Purpose:** Visual budget summary card with progress indicators, status colors, and comprehensive budget health display

**Architecture Overview:**

swift

struct BudgetSummaryCard: View {

let budgeted: Double

let spent: Double

let primaryColor: Color

*// Computed properties with safety checks*

private var progress: Double {

guard budgeted > 0 else { return 0 }

return min(spent / budgeted, 1.0) *// Clamped to 100%*

}

private var percentage: Double { progress \* 100 }

private var isOverBudget: Bool { spent > budgeted }

private var statusColor: Color { isOverBudget ? .red : .green }

}

**Key Visual Components:**

**Three-Section Layout:**

1. **Summary Section:** HStack with budget/spent amounts and percentage circle
2. **Progress Bar:** GeometryReader-based animated progress visualization
3. **Percentage Circle:** 60x60 circular progress indicator with background

**Advanced Progress System:**

swift

*// Animated progress bar*

Rectangle()

.fill(statusColor)

.frame(width: geometry.size.width \* progress, height: 8)

.animation(.easeInOut, value: progress)

*// Circular progress indicator*

Circle()

.trim(from: 0, to: progress)

.stroke(statusColor, lineWidth: 8)

.rotationEffect(.degrees(-90)) *// Start from top*

**Status and Color Logic:**

**Dynamic Status Colors:**

* **Over Budget:** Red color system
* **Under Budget:** Green color system
* **Progress Bar:** Matches status color
* **Percentage Circle:** Coordinated color theming

**Safety and Edge Cases:**

* **Zero Budget:** Progress defaults to 0, prevents division by zero
* **Negative Values:** Handles gracefully with min/max clamping
* **Animation:** Smooth transitions tied to progress value changes

**Integration Points:**

**Data Sources:**

* Receives budget and spent amounts from parent views
* Uses primaryColor from ThemeManager coordination
* Self-contained calculations for progress and status

**Accessibility Features:**

swift

.accessibilityElement(children: .combine)

.accessibilityLabel("Budget summary: \(budgeted.asCurrency) budgeted, \(spent.asCurrency) spent, \(percentage)% used")

**Usage Patterns:**

* Used throughout budget interfaces for consistent summary display
* Provides immediate visual feedback on budget health
* Responsive to budget data changes with smooth animations
* Self-contained component requiring minimal parent coordination

**Performance Considerations:**

* **GeometryReader:** Used efficiently for progress bar width calculations
* **Animation:** Single animation tied to progress value prevents conflicts
* **Computation:** All calculations are lightweight computed properties

**Integration Dependencies:**

* **ThemeManager:** Color coordination through primaryColor parameter
* **Extensions:** Uses Double.asCurrency for formatted display
* **Accessibility:** Follows app-wide accessibility patterns

Both components are designed for **consistent visual presentation** across budget features and integrate seamlessly with the app's theming and accessibility systems.

**27. Features/Budget/Views/Components/MoneyCalculatorView.swift**

**File Type:** Specialized Money Input Interface **Purpose:** Dedicated calculator for monetary amounts with validation and UX optimization

**Architecture:**

swift

public struct MoneyCalculatorView: View {

@Binding private var amount: Double

@State private var inputString = "0"

@State private var isProcessing = false

@State private var hasValidInput = true

@State private var showingConfirmation = false

private enum Constants {

static let buttonSize: CGFloat = 70

static let maxDigits = 12

static let hapticFeedback = UIImpactFeedbackGenerator(style: .light)

}

}

**Advanced Calculator System:**

swift

private enum CalculatorButton: Equatable {

case number(Int), clear, delete

var title: String { */\* number, "C", "⌫" \*/* }

var color: Color { */\* .primary for numbers, .red for actions \*/* }

var accessibilityLabel: String { */\* "Number X", "Clear all", "Delete last digit" \*/* }

}

private let buttonLayout: [[CalculatorButton]] = [

[.number(7), .number(8), .number(9)],

[.number(4), .number(5), .number(6)],

[.number(1), .number(2), .number(3)],

[.clear, .number(0), .delete]

]

**Smart Display System:**

swift

private var formattedDisplay: String {

*// Cent-based calculation (divide by 100)*

*// Smart comma insertion for thousands*

*// $0.00 to $999,999.99 formatting*

*// Real-time validation feedback*

}

private var currentAmount: Double {

guard let value = Double(inputString) else { return 0 }

return value / 100.0 *// Cent-based input system*

}

**UX Enhancements:**

* **Quick Amount Buttons:** $5, $10, $20, $50 shortcuts
* **Haptic Feedback:** Light feedback on button press, error feedback for limits
* **Confirmation Dialog:** For amounts ≥ $100
* **Amount Breakdown:** "X dollars and Y cents" for large amounts
* **Validation:** Real-time against AppConstants.Validation limits
* **Processing Overlay:** Non-blocking save with progress indication

**Accessibility:**

* Complete VoiceOver support with descriptive labels
* Combined accessibility elements
* Action hints for all interactive components

**Integration Points:**

* Used by budget and purchase entry flows
* Integrates with ThemeManager for consistent styling
* Coordinates with ErrorHandler for validation errors
* Supports amount binding for parent view coordination

**28. Features/Budget/Views/Components/BudgetCategoryRow.swift**

**File Type:** Reusable Category Display Component **Purpose:** Standardized category row with progress visualization

**Key Features:**

swift

struct BudgetCategoryRow: View {

let category: String, amount: Double

@EnvironmentObject private var themeManager: ThemeManager

private var categoryLabel: some View *// Color indicator + category name*

private var amountLabel: some View *// Formatted amount display*

private var progressBar: some View *// Visual progress representation*

}

**Visual Design:**

* **Color Indicator:** Small circle with theme-coordinated category color
* **Typography:** Headline font with proper hierarchy
* **Progress Bar:** TODO comment indicates future spent amount integration
* **Accessibility:** Combined element with descriptive labels

**Integration Points:**

* Used in budget views for consistent category display
* Integrates with ThemeManager for color coordination
* Supports navigation to category detail views
* Follows accessibility best practices

**29. Features/Budget/Views/Components/BudgetSummaryCard.swift**

**File Type:** Reusable Summary Display Component **Purpose:** Visual budget summary with progress indicators and status

**Advanced Features:**

swift

struct BudgetSummaryCard: View {

let budgeted: Double, spent: Double, primaryColor: Color

*// Computed properties with safety checks*

private var progress: Double {

guard budgeted > 0 else { return 0 }

return min(spent / budgeted, 1.0) *// Clamped to 100%*

}

private var percentage: Double { progress \* 100 }

private var isOverBudget: Bool { spent > budgeted }

private var statusColor: Color { isOverBudget ? .red : .green }

}

**Visual Components:**

1. **Summary Section:** HStack with budget/spent amounts and percentage circle
2. **Progress Bar:** GeometryReader-based with animated width
3. **Percentage Circle:** ZStack with background + progress circles (60x60 frame)
4. **Accessibility:** Combined element with comprehensive accessibility label

**Animation Features:**

* .easeInOut animation tied to progress value
* Circle rotation at -90 degrees for top start
* 8pt line width with rounded line caps
* Color transitions based on over-budget status

**Integration Points:**

* Used throughout budget interfaces for consistent summary display
* Styled with ThemeManager colors
* Responsive to budget data changes
* Accessible with combined labels and hints

**30. Features/Budget/Views/Components/MoneyCalculatorView.swift**

**File Type:** Specialized Money Input Interface **Purpose:** Dedicated calculator for monetary amounts with validation and UX optimization

**Architecture:**

swift

public struct MoneyCalculatorView: View {

@Binding private var amount: Double

@State private var inputString = "0"

@State private var isProcessing = false

@State private var hasValidInput = true

@State private var showingConfirmation = false

private enum Constants {

static let buttonSize: CGFloat = 70

static let maxDigits = 12

static let hapticFeedback = UIImpactFeedbackGenerator(style: .light)

}

}

**Advanced Calculator System:**

swift

private enum CalculatorButton: Equatable {

case number(Int), clear, delete

var title: String { */\* number, "C", "⌫" \*/* }

var color: Color { */\* .primary for numbers, .red for actions \*/* }

var accessibilityLabel: String { */\* "Number X", "Clear all", "Delete last digit" \*/* }

}

private let buttonLayout: [[CalculatorButton]] = [

[.number(7), .number(8), .number(9)],

[.number(4), .number(5), .number(6)],

[.number(1), .number(2), .number(3)],

[.clear, .number(0), .delete]

]

**Smart Display System:**

swift

private var formattedDisplay: String {

*// Cent-based calculation (divide by 100)*

*// Smart comma insertion for thousands*

*// $0.00 to $999,999.99 formatting*

*// Real-time validation feedback*

}

private var currentAmount: Double {

guard let value = Double(inputString) else { return 0 }

return value / 100.0 *// Cent-based input system*

}

**UX Enhancements:**

* **Quick Amount Buttons:** $5, $10, $20, $50 shortcuts
* **Haptic Feedback:** Light feedback on button press, error feedback for limits
* **Confirmation Dialog:** For amounts ≥ $100
* **Amount Breakdown:** "X dollars and Y cents" for large amounts
* **Validation:** Real-time against AppConstants.Validation limits
* **Processing Overlay:** Non-blocking save with progress indication

**Accessibility:**

* Complete VoiceOver support with descriptive labels
* Combined accessibility elements
* Action hints for all interactive components

**Integration Points:**

* Used by budget and purchase entry flows
* Integrates with ThemeManager for consistent styling
* Coordinates with ErrorHandler for validation errors
* Supports amount binding for parent view coordination

**31. Features/Budget/Views/Components/MoneyCalculatorView.swift**

**File Type:** Specialized Money Input Interface **Purpose:** Dedicated calculator for monetary amounts with validation and UX optimization

**Architecture:**

swift

public struct MoneyCalculatorView: View {

@Binding private var amount: Double

@State private var inputString = "0"

@State private var isProcessing = false

@State private var hasValidInput = true

@State private var showingConfirmation = false

private enum Constants {

static let buttonSize: CGFloat = 70

static let maxDigits = 12

static let hapticFeedback = UIImpactFeedbackGenerator(style: .light)

}

}

**Advanced Calculator System:**

swift

private enum CalculatorButton: Equatable {

case number(Int), clear, delete

var title: String { */\* number, "C", "⌫" \*/* }

var color: Color { */\* .primary for numbers, .red for actions \*/* }

var accessibilityLabel: String { */\* "Number X", "Clear all", "Delete last digit" \*/* }

}

private let buttonLayout: [[CalculatorButton]] = [

[.number(7), .number(8), .number(9)],

[.number(4), .number(5), .number(6)],

[.number(1), .number(2), .number(3)],

[.clear, .number(0), .delete]

]

**Smart Display System:**

swift

private var formattedDisplay: String {

*// Cent-based calculation (divide by 100)*

*// Smart comma insertion for thousands*

*// $0.00 to $999,999.99 formatting*

*// Real-time validation feedback*

}

private var currentAmount: Double {

guard let value = Double(inputString) else { return 0 }

return value / 100.0 *// Cent-based input system*

}

**UX Enhancements:**

* **Quick Amount Buttons:** $5, $10, $20, $50 shortcuts
* **Haptic Feedback:** Light feedback on button press, error feedback for limits
* **Confirmation Dialog:** For amounts ≥ $100
* **Amount Breakdown:** "X dollars and Y cents" for large amounts
* **Validation:** Real-time against AppConstants.Validation limits
* **Processing Overlay:** Non-blocking save with progress indication

**Accessibility:**

* Complete VoiceOver support with descriptive labels
* Combined accessibility elements
* Action hints for all interactive components

**Integration Points:**

* Used by budget and purchase entry flows
* Integrates with ThemeManager for consistent styling
* Coordinates with ErrorHandler for validation errors
* Supports amount binding for parent view coordination

**History Feature (Features/History/)**

**32. Features/History/Views/BudgetHistoryView.swift**

**File Type:** Historical Analysis Interface  
**Purpose:** Comprehensive budget history with charts, filtering, and detailed analytics

**Architecture:**

struct BudgetHistoryView: View {

@EnvironmentObject private var budgetManager: BudgetManager

@EnvironmentObject private var themeManager: ThemeManager

@StateObject private var viewModel = HistoryViewModel()

@State private var selectedTimePeriod: TimePeriod = .thisMonth

@State private var sortOption: BudgetSortOption = .category

@State private var selectedDataPoint: BudgetHistoryData?

}

**Key UI Sections:**

1. **Header Controls:**
   * Time period selector with chip-based navigation
   * Quick stats cards (total budget, spent, remaining)
2. **Chart Section:**
   * Bar chart with budget vs spent comparison
   * Budget limit indicators with red dashed lines
   * Category annotations and interaction support
3. **Chart Legend:**
   * Interactive legend with category selection
   * Color coordination and percentage display
   * Over-budget indicators
4. **Detailed Breakdown:**
   * List view of all categories with comprehensive data
   * Context menus for category actions
   * Navigation to detailed category views

**Chart Configuration:**

private let chartColors: [Color] = [

Color(r: 0.12, g: 0.58, b: 0.95), // Blue

Color(r: 0.99, g: 0.85, b: 0.21), // Yellow

Color(r: 0.18, g: 0.80, b: 0.44), // Green

// ... 7 total colors

]

private let maxDataPoints = 100

private let chartAnimationDuration: Double = 0.8

**Advanced Features:**

1. **Export System:**
   * Multiple format support (CSV, JSON, PDF)
   * Configurable export options
   * Progress tracking for large exports
   * Share sheet integration
2. **Category Detail Sheets:**
   * Complete category analysis with metrics
   * Progress visualization and insights
   * Action buttons for editing and navigation
3. **Filter and Sort:**
   * Comprehensive filtering options
   * Multiple sort criteria with direction
   * Filter preview and validation

**Integration Points:**

* Uses HistoryViewModel for comprehensive data management
* Integrates with FilterSortView for advanced filtering
* Coordinates with ExportOptionsView for data export
* Publishes navigation events for app coordination

**33. Features/History/ViewModels/HistoryViewModel.swift**

**File Type:** History Business Logic  
**Purpose:** Advanced history management with filtering, analytics, and export capabilities

**Architecture:**

@MainActor

public final class HistoryViewModel: ObservableObject {

@Published public private(set) var viewState: ViewState = .loading

@Published public private(set) var budgetHistoryData: [BudgetHistoryData] = []

@Published public private(set) var filteredData: [BudgetHistoryData] = []

@Published public private(set) var analyticsData: AnalyticsData?

@Published public var filterConfiguration: FilterConfiguration = FilterConfiguration()

}

**Advanced Analytics:**

public struct AnalyticsData {

let totalBudgeted: Double, totalSpent: Double, totalRemaining: Double

let averageSpentPerCategory: Double, categoriesOverBudget: Int, categoriesUnderBudget: Int

let biggestOverspend: (category: String, amount: Double)?

let biggestUnderspend: (category: String, amount: Double)?

let spendingTrend: SpendingTrend, efficiencyScore: Double

var overallHealth: BudgetHealth // excellent/good/warning/poor

}

**Filtering System:**

public struct FilterConfiguration {

var timePeriod: TimePeriod, selectedCategories: Set<String>

var sortOption: BudgetSortOption, sortAscending: Bool

var showOnlyOverBudget: Bool, minimumAmount: Double?, maximumAmount: Double?

var hasActiveFilters: Bool, filterDescription: String

}

**Chart Integration:**

public func getConfiguredChartData(for type: ChartType) -> ChartConfiguration

public func getLegendData() -> [LegendItem]

public func getAvailableFilterOptions() -> FilterOptions

public func applyQuickFilter(\_ filter: QuickFilter)

**Export Capabilities:**

public func exportData(format: ExportFormat = .csv) async throws -> URL

private func exportToCSV(\_ data: [ExportDataRow]) async throws -> URL

private func exportToJSON(\_ data: [ExportDataRow]) async throws -> URL

**Performance Features:**

* **Caching System:** Filter results cached with 5-minute validity
* **Background Processing:** Heavy calculations performed off-main-thread
* **Debounced Updates:** Filter changes debounced to prevent excessive processing
* **Memory Management:** Automatic cleanup of old data and cache entries

**Integration Points:**

* Manages all historical data analysis and presentation
* Coordinates with BudgetManager for data source
* Provides comprehensive analytics for UI display
* Handles complex filtering and export operations

**34. Features/History/Views/Components/BudgetHistoryRow.swift**

**File Type:** History Item Display Component  
**Purpose:** Detailed budget history row with progress visualization and status

**Key Features:**

swift

struct BudgetHistoryRow: View {

let data: BudgetHistoryData, color: Color

private var categoryAndAmounts: some View *// Budget vs spent display*

private var progressBar: some View *// Visual progress with color coding*

private var footerInfo: some View *// Remaining amount and percentage*

}

**Visual Design:**

* **Dual Amount Display:** Budget and spent amounts with color coding
* **Progress Visualization:** Width-based progress bar with status colors
* **Status Indicators:** Over-budget highlighting with red colors
* **Comprehensive Footer:** Remaining amount and usage percentage

**Accessibility:**

* **Combined Element:** Single accessibility element with comprehensive description
* **Status Announcements:** Clear over/under budget status
* **Value Descriptions:** All monetary values properly labeled

**Integration Points:**

* Used in history views for consistent data display
* Receives BudgetHistoryData from view models
* Styled with coordinated colors and theme integration
* Supports interaction for detailed views

# **Section 35: Features/History/Views/Components/ChartLegend.swift**

**File Type:** Chart Legend Components  
**Purpose:** Interactive chart legends with selection support and accessibility for data visualization

## **Architecture Overview:**

swift

*/// Individual legend item for chart visualizations*

struct ChartLegend: View {

let data: BudgetHistoryData

let color: Color

private var formattedPercentage: String {

data.percentageSpent.formatted(.percent.precision(.fractionLength(1)))

}

}

*/// Group of chart legends with header and organization*

struct ChartLegendGroup: View {

let data: [BudgetHistoryData]

let colors: [Color]

*// Computed analytics*

private var totalSpent: Double

private var totalBudget: Double

}

## **Key Components:**

### **Individual Legend (ChartLegend):**

* **Color Indicator:** 12x12 rectangle with 2pt corner radius
* **Category Display:** Text with line limit and percentage
* **Status Integration:** Red/green color coding for over/under budget
* **Interactive:** Supports selection and accessibility

### **Legend Group (ChartLegendGroup):**

* **Header Section:** "Categories" title with category count
* **Enumerated Display:** Color coordination with chart data
* **Background Styling:** Gray background with corner radius
* **Combined Accessibility:** Single element with comprehensive description

## **Visual Design System:**

### **Layout Structure:**

swift

HStack(spacing: 8) {

*// Color Indicator*

Rectangle()

.fill(color)

.frame(width: 12, height: 12)

.cornerRadius(2)

*// Category Name*

Text(data.category)

.font(.caption)

.lineLimit(1)

Spacer()

*// Percentage with status color*

Text(formattedPercentage)

.font(.caption)

.foregroundColor(data.isOverBudget ? .red : .green)

.bold()

}

### **Color Coordination:**

* **Index-Based Colors:** colors[index % colors.count] for cycling
* **Status Colors:** Over-budget indicators in red
* **Theme Integration:** Background uses Color.gray.opacity(0.05)

## **Integration Points:**

### **Data Dependencies:**

* **BudgetHistoryData:** Requires category, percentage, over-budget status
* **Color Array:** External color coordination with chart components
* **Accessibility:** Combined labels for screen reader support

### **Chart Coordination:**

* Used by chart components for legend display
* Color array must match chart slice colors
* Selection state coordination with parent charts

## **Accessibility Features:**

swift

.accessibilityElement(children: .combine)

.accessibilityLabel("Chart legend showing spending percentages for each category")

.accessibilityLabel("\(data.category): \(formattedPercentage) of budget spent")

# **Section 36: Features/History/Views/Components/FilterSortView.swift**

**File Type:** Advanced Filtering Interface  
**Purpose:** Comprehensive filtering and sorting with live preview, validation, and persistence

## **Architecture Overview:**

swift

@MainActor

class FilterSortViewModel: ObservableObject {

@Published var groupByCategory = false

@Published var showOnlyOverBudget = false

@Published var showOnlyWithActivity = false

@Published var showZeroBudget = true

@Published var showHighSpending = false

@Published var showPercentages = true

@Published var showTrends = false

@Published var compactView = false

@Published var previewText: String?

}

## **Filter Categories:**

### **Data Filters:**

* **Over Budget Only:** showOnlyOverBudget - Filters to categories exceeding budget
* **Activity Filter:** showOnlyWithActivity - Shows only categories with transactions
* **Zero Budget:** showZeroBudget - Include/exclude categories with no budget set
* **High Spending:** showHighSpending - Categories >75% of budget or above average

### **Display Options:**

* **Grouping:** groupByCategory - Organizational display options
* **Percentages:** showPercentages - Toggle percentage display
* **Trends:** showTrends - Show trend indicators
* **Compact View:** compactView - Condensed display mode

### **Validation System:**

swift

struct ValidationResult {

let isValid: Bool

let warnings: [String]

var hasWarnings: Bool { !warnings.isEmpty }

var summary: String *// User-friendly validation summary*

}

func validateConfiguration() -> ValidationResult {

*// Detects conflicting filter combinations*

*// Warns about filters that may show no results*

*// Provides actionable feedback*

}

## **Advanced Features:**

### **Live Preview System:**

* **Real-time Preview:** previewText updates with filter combinations
* **Natural Language:** Generates human-readable filter descriptions
* **Conflict Detection:** Warns about incompatible filter combinations

### **Persistence Integration:**

swift

private enum UserDefaultsKeys {

static let groupByCategory = "FilterSort.groupByCategory"

static let showOnlyOverBudget = "FilterSort.showOnlyOverBudget"

*// ... comprehensive key management*

}

private func setupPropertyObservers() {

*// Automatic saving with Combine publishers*

*// Debounced updates to prevent excessive saves*

}

### **Filter Application Logic:**

swift

struct FilterCriteria {

func apply(to data: [BudgetHistoryData]) -> [BudgetHistoryData] {

*// Multi-stage filtering pipeline*

*// Performance-optimized filter chain*

*// High spending calculation with average comparison*

}

func getDisplayConfig() -> DisplayConfiguration {

*// Coordinates display options with filter state*

}

}

## **Integration Points:**

### **Data Processing:**

* **Input:** [BudgetHistoryData] from HistoryViewModel
* **Output:** Filtered and configured data for display
* **Performance:** Optimized filter chain for large datasets

### **UI Coordination:**

* **Sheet Presentation:** Modal interface from history views
* **Live Feedback:** Real-time preview and validation
* **Settings Persistence:** User preferences maintained across sessions

### **Error Handling:**

* **Validation Warnings:** Non-blocking user guidance
* **Conflict Resolution:** Suggests filter combination improvements
* **Graceful Degradation:** Handles edge cases without crashes

## **Configuration Export/Import:**

swift

func exportConfiguration() -> [String: Any] {

*// Complete filter state serialization*

*// Version tracking for migration*

*// Metadata inclusion for debugging*

}

func importConfiguration(\_ data: [String: Any]) {

*// Safe configuration restoration*

*// Validation before application*

*// Error handling for corrupted data*

}

# **Section 37: Features/Onboarding/Views/WelcomePopupView.swift**

**File Type:** First-Launch Experience  
**Purpose:** Comprehensive onboarding with personalization, system integration, and accessibility

## **Architecture Overview:**

swift

struct WelcomePopupView: View {

@EnvironmentObject private var settingsManager: SettingsManager

@EnvironmentObject private var themeManager: ThemeManager

@EnvironmentObject private var errorHandler: ErrorHandler

@Binding var isPresented: Bool

@State private var currentStep: OnboardingStep = .welcome

@State private var name: String = ""

@State private var selectedCurrency: String = "USD"

@State private var selectedThemeColor: ThemeManager.ColorOption

}

## **Onboarding Flow System:**

### **Step Progression:**

swift

private enum OnboardingStep: Int, CaseIterable {

case welcome = 0, personalization = 1, preferences = 2,

notifications = 3, completion = 4

var progressValue: Double { Double(rawValue) / Double(allCases.count - 1) }

var title: String, subtitle: String, icon: String

}

### **Step-by-Step Breakdown:**

1. **Welcome (0):** Feature highlights with animated icons
2. **Personalization (1):** Name input and currency selection with validation
3. **Preferences (2):** Theme color picker and haptic feedback toggle
4. **Notifications (3):** System permission integration with graceful fallback
5. **Completion (4):** Success animation and configuration summary

## **Advanced UX Features:**

### **Animation System:**

* **Spring Animations:** 0.6s duration for step transitions
* **Staggered Appearances:** Element-by-element animation entrance
* **Symbol Effects:** Icon bounce effects for visual feedback
* **Theme Transitions:** Live preview during color selection

### **Validation and Error Handling:**

swift

private func validateAndSavePersonalization() async throws {

let trimmedName = name.trimmingCharacters(in: .whitespacesAndNewlines)

guard !trimmedName.isEmpty else {

throw AppError.validation(message: "Please enter your name")

}

guard trimmedName.count >= 2 && trimmedName.count <= maxNameLength else {

throw AppError.validation(message: "Name must be 2-50 characters")

}

*// Safe settings updates with error recovery*

}

### **System Integration:**

* **Notification Permissions:** UNUserNotificationCenter integration
* **Settings Coordination:** SettingsManager updates with validation
* **Theme Application:** Live ThemeManager updates
* **Graceful Fallbacks:** Handles permission denials elegantly

## **Accessibility Excellence:**

### **Focus Management:**

swift

@AccessibilityFocusState private var focusedField: FormField?

private func updateFocusForCurrentStep() {

DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {

switch currentStep {

case .personalization: focusedField = .nameField

case .preferences: focusedField = .themePicker

case .notifications: focusedField = .notificationToggle

default: focusedField = nil

}

}

}

### **Dynamic Type Support:**

* **Size Adaptation:** Responsive to accessibility text sizes
* **Layout Adjustments:** Content spacing adapts to text size
* **Icon Scaling:** Dynamic icon sizes for accessibility

### **VoiceOver Integration:**

* **Step Announcements:** Automatic step change announcements
* **Comprehensive Labels:** Descriptive accessibility labels
* **Navigation Hints:** Clear interaction guidance

## **Integration Points:**

### **Manager Coordination:**

* **SettingsManager:** User preferences, currency, notification settings
* **ThemeManager:** Color selection with live preview
* **NotificationManager:** Permission requests and scheduling
* **ErrorHandler:** Centralized error presentation

### **Data Flow:**

User Input → Validation → Manager Updates → System Integration → Completion

↓ ↓ ↓ ↓ ↓

Error Display ← Retry Logic ← Change Detection ← Permission ← Success

### **State Management:**

* **Progress Tracking:** Linear progress with step indicators
* **Completion Validation:** Each step validates before progression
* **Error Recovery:** Step-specific error handling with retry options

# Section 38: Features/Purchases/Views/Components/TransactionRowView.swift

**File Type:** Enhanced Transaction Display Component  
**Purpose:** Standardized transaction row with comprehensive accessibility, theming, and sophisticated category visualization

## Architecture Overview:

swift

struct TransactionRowView: View {

*// MARK: - Properties*

let entry: BudgetEntry

@EnvironmentObject private var themeManager: ThemeManager

*// MARK: - Computed Properties*

private var formattedAmount: String {

entry.amount.asCurrency

}

private var iconLetter: String {

String(entry.category.prefix(1).uppercased())

}

private var formattedDate: String {

entry.date.formatted(date: .abbreviated, time: .omitted)

}

}

## Visual Design System:

### Three-Section Layout Implementation:

swift

var body: some View {

HStack(spacing: 16) {

categoryBadge

VStack(alignment: .leading, spacing: 4) {

categoryAndAmount

dateAndNote

}

}

.padding(.vertical, 8)

.accessibilityElement(children: .combine)

.accessibilityLabel(createAccessibilityLabel())

.accessibilityHint("Double tap to edit transaction")

}

### Category Badge with Color Coordination:

swift

private var categoryBadge: some View {

ZStack {

Circle()

.fill(themeManager.primaryColor.opacity(0.1))

.frame(width: 40, height: 40)

Text(iconLetter)

.font(.headline)

.foregroundColor(themeManager.primaryColor)

}

.accessibilityHidden(true)

}

### Content Structure with Typography Hierarchy:

swift

private var categoryAndAmount: some View {

HStack {

Text(entry.category)

.font(.headline)

.lineLimit(1)

Spacer()

Text(formattedAmount)

.font(.headline)

.foregroundColor(themeManager.primaryColor)

}

}

private var dateAndNote: some View {

HStack {

Text(formattedDate)

.font(.caption)

.foregroundColor(.secondary)

if let note = entry.note {

Text("•")

.font(.caption)

.foregroundColor(.secondary)

Text(note)

.font(.caption)

.foregroundColor(.secondary)

.lineLimit(1)

}

}

}

## Enhanced Accessibility Implementation:

### Comprehensive VoiceOver Support:

swift

private func createAccessibilityLabel() -> String {

var label = "\(entry.category) transaction for \(formattedAmount) on \(formattedDate)"

if let note = entry.note {

label += ". Note: \(note)"

}

return label

}

### Accessibility Features:

* **Combined Element:** Single accessibility element for entire row
* **Descriptive Labels:** Complete transaction information in natural language
* **Action Hints:** Clear interaction guidance ("Double tap to edit transaction")
* **Hidden Elements:** Category badge hidden to prevent redundant information
* **Context Information:** Date, amount, category, and optional note all included

## Advanced Preview System:

### Multiple Testing Scenarios:

swift

#if DEBUG

struct TransactionRowView\_Previews: PreviewProvider {

static var previews: some View {

Group {

*// Standard Entry*

TransactionRowView(

entry: BudgetEntry.mock(

amount: 42.50,

category: "Groceries",

date: Date(),

note: "Weekly shopping"

)

)

.environmentObject(ThemeManager.shared)

.previewLayout(.sizeThatFits)

.padding()

.previewDisplayName("Standard Entry")

*// Entry without Note*

TransactionRowView(

entry: BudgetEntry.mock(

amount: 99.99,

category: "Entertainment",

date: Date()

)

)

.environmentObject(ThemeManager.shared)

.previewLayout(.sizeThatFits)

.padding()

.previewDisplayName("No Note")

*// Long Category and Note*

TransactionRowView(

entry: BudgetEntry.mock(

amount: 150.00,

category: "Very Long Category Name That Should Truncate",

date: Date(),

note: "This is a very long note that should be truncated after a certain length"

)

)

.environmentObject(ThemeManager.shared)

.previewLayout(.sizeThatFits)

.padding()

.previewDisplayName("Long Content")

*// Dark Mode*

TransactionRowView(

entry: BudgetEntry.mock(

amount: 75.00,

category: "Transportation",

date: Date(),

note: "Bus fare"

)

)

.environmentObject(ThemeManager.shared)

.preferredColorScheme(.dark)

.previewLayout(.sizeThatFits)

.padding()

.previewDisplayName("Dark Mode")

*// List Context*

List {

TransactionRowView(

entry: BudgetEntry.mock(

amount: 42.50,

category: "Groceries",

date: Date(),

note: "Weekly shopping"

)

)

TransactionRowView(

entry: BudgetEntry.mock(

amount: 99.99,

category: "Entertainment",

date: Date()

)

)

}

.environmentObject(ThemeManager.shared)

.previewDisplayName("List Context")

}

}

}

#endif

## Integration Points:

### Data Dependencies:

* **BudgetEntry:** Complete transaction data structure with validation
* **ThemeManager:** Color coordination and consistent visual theming
* **DateExtensions:** Formatted date display with localization
* **NumberFormatterExtensions:** Currency formatting with locale support

### Usage Patterns:

* **PurchasesView:** Primary transaction listing interface
* **OverviewView:** Recent transactions display
* **HistoryView:** Historical transaction browsing
* **Search Results:** Filtered transaction display

### Theme Integration:

* **Primary Color:** Category badge background and amount text
* **Semantic Colors:** Uses theme-aware secondary and tertiary text colors
* **Consistent Styling:** Follows app-wide design patterns and spacing

## Performance Considerations:

### Efficient Rendering:

* **Lightweight Calculations:** All computed properties are simple transformations
* **Minimal State:** No internal state management for better performance
* **Text Caching:** Formatted strings computed once per render
* **Layout Optimization:** Simple HStack/VStack structure for fast layout

### Memory Management:

* **No Retained References:** No strong references to external objects
* **Environment Objects:** Proper environment object usage
* **Reusable Design:** Stateless component suitable for list virtualization

This component serves as the **foundation for transaction display** throughout the app, ensuring consistent presentation, accessibility, and performance across all transaction-related interfaces.

# Section 39: Features/Purchases/Views/PurchasesView.swift

**File Type:** Comprehensive Purchase Management Interface  
**Purpose:** Advanced purchase listing with filtering, search, state management, and sophisticated error handling

## Architecture Overview:

swift

struct PurchasesView: View {

*// MARK: - Environment*

@EnvironmentObject private var budgetManager: BudgetManager

@EnvironmentObject private var themeManager: ThemeManager

@EnvironmentObject private var settingsManager: SettingsManager

@EnvironmentObject private var errorHandler: ErrorHandler

@Environment(\.dismiss) private var dismiss

*// MARK: - State*

@State private var searchText = ""

@State private var selectedTimePeriod: TimePeriod = .thisMonth

@State private var selectedCategory: String = "All"

@State private var sortOption: BudgetSortOption = .date

@State private var sortAscending = false

@State private var filteredEntries: [BudgetEntry] = []

@State private var loadingState: LoadingState = .idle

@State private var lastRefreshDate: Date?

}

## Advanced State Management:

### Loading State System:

swift

private enum LoadingState {

case idle

case loading

case loaded

case error(AppError)

case refreshing

var isLoading: Bool {

switch self {

case .loading, .refreshing: return true

default: return false

}

}

var hasError: Bool {

if case .error = self { return true }

return false

}

}

### Filter State Management:

swift

private var hasActiveFilters: Bool {

return selectedTimePeriod != .thisMonth ||

selectedCategory != "All" ||

!searchText.isEmpty ||

sortOption != .date ||

sortAscending != false

}

private var availableCategories: [String] {

let allCategories = budgetManager.getAvailableCategories()

return ["All"] + allCategories.sorted()

}

## Comprehensive UI Implementation:

### Navigation Toolbar with Context-Sensitive Actions:

swift

.toolbar {

ToolbarItemGroup(placement: .navigationBarTrailing) {

HStack {

*// Export button*

Button {

showingExportOptions = true

} label: {

Image(systemName: "square.and.arrow.up")

}

.disabled(!canExport)

*// Filter button with active state indication*

Button {

showingFilterOptions = true

} label: {

Image(systemName: hasActiveFilters ? "line.3.horizontal.decrease.circle.fill" : "line.3.horizontal.decrease.circle")

.foregroundColor(hasActiveFilters ? themeManager.primaryColor : .primary)

}

*// Add purchase button*

Button {

showingAddPurchase = true

} label: {

Image(systemName: "plus.circle.fill")

.foregroundColor(themeManager.primaryColor)

}

}

}

ToolbarItem(placement: .navigationBarLeading) {

if isRefreshing {

ProgressView()

.scaleEffect(0.8)

}

}

}

### Summary Header with Statistics:

swift

private var summaryHeader: some View {

HStack {

VStack(alignment: .leading, spacing: 4) {

Text("\(entryCount) Purchase\(entryCount == 1 ? "" : "s")")

.font(.headline)

.foregroundColor(.primary)

Text("Total: \(totalAmount.asCurrency)")

.font(.subheadline)

.foregroundColor(themeManager.primaryColor)

.fontWeight(.semibold)

}

Spacer()

if let lastRefresh = lastRefreshDate {

VStack(alignment: .trailing, spacing: 2) {

Text("Updated")

.font(.caption2)

.foregroundColor(.secondary)

Text(formatRelativeTime(lastRefresh))

.font(.caption2)

.foregroundColor(.secondary)

}

}

}

.padding()

.background(Color(.secondarySystemGroupedBackground))

.cornerRadius(12)

}

## Advanced Filtering and Search System:

### Debounced Filter Updates:

swift

private let filterDebounceInterval: TimeInterval = 0.3

private var lastFilterUpdate = Date()

private func scheduleFilterUpdate() {

*// Cancel previous filter update*

let updateTime = Date()

lastFilterUpdate = updateTime

Task {

*// Debounce the update*

try? await Task.sleep(nanoseconds: UInt64(filterDebounceInterval \* 1\_000\_000\_000))

*// Only proceed if this is still the latest update*

if lastFilterUpdate == updateTime {

await filterEntries()

}

}

}

### Multi-Criteria Filtering:

swift

private func filterEntries() async {

do {

let allEntries = try await budgetManager.getEntries(

for: selectedTimePeriod,

category: selectedCategory == "All" ? nil : selectedCategory,

sortedBy: sortOption,

ascending: sortAscending

)

var filtered = allEntries

*// Apply search filter if needed*

if !searchText.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty {

let searchTerms = searchText.lowercased()

filtered = filtered.filter { entry in

entry.category.lowercased().contains(searchTerms) ||

(entry.note?.lowercased().contains(searchTerms) ?? false) ||

entry.formattedAmount.contains(searchTerms)

}

}

await MainActor.run {

filteredEntries = filtered

lastFilterUpdate = Date()

}

} catch {

await handleLoadError(AppError.from(error))

}

}

## Enhanced Transaction Display:

### Transaction List with Context Menus:

swift

private var purchasesList: some View {

List {

*// Error banner (if any)*

if let error = localError {

Section {

InlineErrorView(

error: error,

onDismiss: { clearLocalError() },

onRetry: error.isRetryable ? {

Task { await performRetry() }

} : nil

)

}

.listRowInsets(EdgeInsets())

.listRowSeparator(.hidden)

}

*// Purchases with enhanced interaction*

ForEach(displayedEntries) { entry in

TransactionRowView(

entry: entry,

onTap: {

selectedEntry = entry

showingUpdatePurchase = true

},

onDelete: {

Task {

await deletePurchase(entry)

}

}

)

.listRowSeparator(.hidden)

.listRowInsets(EdgeInsets(top: 4, leading: 16, bottom: 4, trailing: 16))

}

.onDelete(perform: deleteEntries)

}

.listStyle(.plain)

.animation(.easeInOut(duration: 0.3), value: displayedEntries.count)

}

## Sophisticated Empty State Handling:

### Context-Aware Empty States:

swift

private var emptyStateView: some View {

VStack(spacing: 24) {

Image(systemName: "cart")

.font(.system(size: 64))

.foregroundColor(.gray)

VStack(spacing: 12) {

Text("No Purchases Found")

.font(.title2)

.fontWeight(.semibold)

if hasActiveFilters {

Text("Try adjusting your filters or search terms")

.font(.subheadline)

.foregroundColor(.secondary)

.multilineTextAlignment(.center)

Button("Clear Filters") {

clearAllFilters()

}

.buttonStyle(.borderedProminent)

.tint(themeManager.primaryColor)

} else {

Text("Add your first purchase to get started")

.font(.subheadline)

.foregroundColor(.secondary)

.multilineTextAlignment(.center)

Button("Add Purchase") {

showingAddPurchase = true

}

.buttonStyle(.borderedProminent)

.tint(themeManager.primaryColor)

}

}

}

.frame(maxWidth: .infinity, maxHeight: .infinity)

.padding()

}

## Advanced Data Operations:

### Purchase Management with Error Handling:

swift

private func deletePurchase(\_ entry: BudgetEntry) async {

do {

try await budgetManager.deleteEntry(entry)

*// Update local state*

await MainActor.run {

filteredEntries.removeAll { $0.id == entry.id }

*// Show success feedback*

if settingsManager.enableHapticFeedback {

let impactFeedback = UIImpactFeedbackGenerator(style: .light)

impactFeedback.impactOccurred()

}

}

} catch {

await MainActor.run {

localError = AppError.from(error)

showingErrorDetails = true

}

}

}

### Refresh System with Performance Monitoring:

swift

private func performRefresh() async {

await MainActor.run {

isRefreshing = true

loadingState = .refreshing

}

*// Add slight delay for better UX*

try? await Task.sleep(nanoseconds: 500\_000\_000) *// 0.5 seconds*

*// Reload data from budget manager*

budgetManager.loadData()

*// Re-filter entries*

await filterEntries()

await MainActor.run {

isRefreshing = false

loadingState = .loaded

lastRefreshDate = Date()

}

}

## Integration Points:

### Manager Dependencies:

* **BudgetManager:** Primary data source with real-time updates
* **ThemeManager:** Consistent styling and color coordination
* **SettingsManager:** User preferences and haptic feedback
* **ErrorHandler:** Centralized error presentation and recovery

### Sheet Presentations:

* **PurchaseEntryView:** New purchase creation
* **UpdatePurchaseView:** Purchase modification
* **FilterOptionsView:** Advanced filtering interface
* **ExportOptionsView:** Data export configuration

### Performance Features:

* **Debounced Filtering:** Prevents excessive recomputation
* **List Virtualization:** Efficient rendering for large datasets
* **Background Processing:** Heavy operations off main thread
* **Memory Management:** Proper cleanup and resource management

This view represents a **comprehensive purchase management interface** that demonstrates advanced SwiftUI patterns, sophisticated state management, and enterprise-level error handling used throughout the application.

# Section 40: Features/Purchases/ViewModels/PurchasesViewModel.swift

**File Type:** Advanced Purchase Management Logic  
**Purpose:** Comprehensive purchase state management with filtering, analytics, performance monitoring, and sophisticated error handling

## Architecture Overview:

swift

@MainActor

public final class PurchasesViewModel: ObservableObject {

*// MARK: - Dependencies*

private let budgetManager: BudgetManager

private let errorHandler: ErrorHandler

private let settingsManager: SettingsManager

*// MARK: - Published Properties*

@Published public private(set) var loadingState: LoadingState = .idle

@Published public private(set) var allEntries: [BudgetEntry] = []

@Published public private(set) var filteredEntries: [BudgetEntry] = []

@Published public private(set) var availableCategories: [String] = []

@Published public private(set) var statistics: PurchaseStatistics?

@Published public private(set) var lastRefreshDate: Date?

@Published public private(set) var currentOperation: OperationType?

*// MARK: - Filter Properties with Real-time Updates*

@Published public var searchText: String = "" {

didSet { scheduleFilterUpdate() }

}

@Published public var selectedCategory: String = "All" {

didSet { scheduleFilterUpdate() }

}

@Published public var selectedTimePeriod: TimePeriod = .thisMonth {

didSet { scheduleFilterUpdate() }

}

@Published public var sortOption: BudgetSortOption = .date {

didSet { scheduleFilterUpdate() }

}

@Published public var sortAscending: Bool = false {

didSet { scheduleFilterUpdate() }

}

}

## Advanced State Management System:

### Loading State with Error Context:

swift

public enum LoadingState: Equatable {

case idle

case loading

case loaded

case refreshing

case error(AppError)

public var isLoading: Bool {

switch self {

case .loading, .refreshing: return true

default: return false

}

}

public var hasError: Bool {

if case .error = self { return true }

return false

}

public var errorValue: AppError? {

if case .error(let error) = self { return error }

return nil

}

public static func == (lhs: LoadingState, rhs: LoadingState) -> Bool {

switch (lhs, rhs) {

case (.idle, .idle), (.loading, .loading), (.loaded, .loaded), (.refreshing, .refreshing):

return true

case (.error(let lhsError), .error(let rhsError)):

return lhsError.id == rhsError.id

default:

return false

}

}

}

### Operation Type Tracking:

swift

public enum OperationType {

case loading

case adding

case updating(BudgetEntry)

case deleting(BudgetEntry)

case exporting

case importing

}

### Advanced Filter State Management:

swift

public enum FilterState {

case none

case search(String)

case category(String)

case timePeriod(TimePeriod)

case combined(search: String?, category: String?, timePeriod: TimePeriod?)

public var hasActiveFilters: Bool {

switch self {

case .none:

return false

case .search(let text):

return !text.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty

case .category(let category):

return category != "All"

case .timePeriod(let period):

return period != .thisMonth

case .combined(let search, let category, let timePeriod):

return (search?.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty == false) ||

(category != "All" && category != nil) ||

(timePeriod != .thisMonth && timePeriod != nil)

}

}

}

## Comprehensive Analytics System:

### Purchase Statistics:

swift

public struct PurchaseStatistics {

public let totalAmount: Double

public let entryCount: Int

public let averageAmount: Double

public let categoryBreakdown: [String: Double]

public let largestPurchase: BudgetEntry?

public let smallestPurchase: BudgetEntry?

public let mostFrequentCategory: String?

public let dateRange: DateInterval?

public var isEmpty: Bool {

return entryCount == 0

}

public var formattedTotalAmount: String {

return NumberFormatter.formatCurrency(totalAmount)

}

public var formattedAverageAmount: String {

return NumberFormatter.formatCurrency(averageAmount)

}

}

### Advanced Analytics Methods:

swift

*/// Get spending trends by month*

public func getSpendingTrends() -> [String: Double] {

let calendar = Calendar.current

let grouped = Dictionary(grouping: filteredEntries) { entry in

calendar.dateInterval(of: .month, for: entry.date)?.start ?? entry.date

}

return grouped.mapValues { entries in

entries.reduce(0) { $0 + $1.amount }

}

}

*/// Get category insights with percentages*

public func getCategoryInsights() -> [(category: String, amount: Double, percentage: Double)] {

guard let stats = statistics, stats.totalAmount > 0 else { return [] }

return stats.categoryBreakdown.map { category, amount in

let percentage = (amount / stats.totalAmount) \* 100

return (category: category, amount: amount, percentage: percentage)

}.sorted { $0.amount > $1.amount }

}

*/// Get recent activity summary*

public func getRecentActivitySummary(days: Int = 7) -> (count: Int, amount: Double) {

let cutoffDate = Calendar.current.date(byAdding: .day, value: -days, to: Date()) ?? Date()

let recentEntries = filteredEntries.filter { $0.date >= cutoffDate }

return (

count: recentEntries.count,

amount: recentEntries.reduce(0) { $0 + $1.amount }

)

}

## Performance Monitoring and Caching:

### Advanced Caching System:

swift

private var lastFilterUpdate = Date()

private let filterDebounceInterval: TimeInterval = 0.3

private var filterUpdateTask: Task<Void, Never>?

private var statisticsCache: (entries: [BudgetEntry], stats: PurchaseStatistics)?

private func scheduleFilterUpdate() {

*// Cancel previous filter update*

filterUpdateTask?.cancel()

let updateTime = Date()

lastFilterUpdate = updateTime

filterUpdateTask = Task {

*// Debounce the update*

try? await Task.sleep(nanoseconds: UInt64(filterDebounceInterval \* 1\_000\_000\_000))

*// Only proceed if this is still the latest update*

if lastFilterUpdate == updateTime && !Task.isCancelled {

await applyFiltersAndSort()

}

}

}

### Performance Metrics:

swift

private var operationMetrics: [String: TimeInterval] = [:]

private let metricsQueue = DispatchQueue(label: "com.brandonsbudget.purchases.metrics", qos: .utility)

private func recordMetric(\_ operation: String, duration: TimeInterval) {

metricsQueue.async {

self.operationMetrics[operation] = duration

#if DEBUG

if duration > 1.0 {

print("⚠️ PurchasesViewModel: Slow operation '\(operation)' took \(String(format: "%.2f", duration \* 1000))ms")

}

#endif

}

}

## Advanced Data Operations:

### Sophisticated Filtering and Sorting:

swift

private func applyFiltersAndSort() async {

let startTime = Date()

var filtered = allEntries

*// Apply time period filter*

if selectedTimePeriod != .thisMonth {

let interval = selectedTimePeriod.dateInterval()

filtered = filtered.filter { entry in

entry.date >= interval.start && entry.date <= interval.end

}

}

*// Apply category filter*

if selectedCategory != "All" {

filtered = filtered.filter { $0.category == selectedCategory }

}

*// Apply search filter*

let trimmedSearch = searchText.trimmingCharacters(in: .whitespacesAndNewlines)

if !trimmedSearch.isEmpty {

let searchTerms = trimmedSearch.lowercased()

filtered = filtered.filter { entry in

entry.category.lowercased().contains(searchTerms) ||

(entry.note?.lowercased().contains(searchTerms) ?? false) ||

entry.formattedAmount.contains(searchTerms)

}

}

*// Apply sorting*

filtered = sortEntries(filtered, by: sortOption, ascending: sortAscending)

*// Update filtered entries*

filteredEntries = filtered

*// Update statistics*

statistics = calculateStatistics(for: filtered)

recordMetric("applyFiltersAndSort", duration: Date().timeIntervalSince(startTime))

}

private func sortEntries(

\_ entries: [BudgetEntry],

by option: BudgetSortOption,

ascending: Bool

) -> [BudgetEntry] {

let sorted = entries.sorted { entry1, entry2 in

let result: Bool

switch option {

case .date:

result = entry1.date < entry2.date

case .amount, .amountSpent:

result = entry1.amount < entry2.amount

case .category:

result = entry1.category < entry2.category

case .budgetedAmount:

result = entry1.amount < entry2.amount *// Fallback to amount*

}

return ascending ? result : !result

}

return sorted

}

### Statistics Calculation with Caching:

swift

private func calculateStatistics(for entries: [BudgetEntry]) -> PurchaseStatistics {

*// Use cache if entries haven't changed*

if let cache = statisticsCache, cache.entries.count == entries.count {

let entriesMatch = zip(cache.entries, entries).allSatisfy { $0.id == $1.id }

if entriesMatch {

return cache.stats

}

}

let totalAmount = entries.reduce(0) { $0 + $1.amount }

let entryCount = entries.count

let averageAmount = entryCount > 0 ? totalAmount / Double(entryCount) : 0

*// Category breakdown*

let categoryBreakdown = Dictionary(grouping: entries, by: { $0.category })

.mapValues { categoryEntries in

categoryEntries.reduce(0) { $0 + $1.amount }

}

*// Find largest and smallest purchases*

let sortedByAmount = entries.sorted { $0.amount > $1.amount }

let largestPurchase = sortedByAmount.first

let smallestPurchase = sortedByAmount.last

*// Most frequent category*

let mostFrequentCategory = categoryBreakdown.max(by: { $0.value < $1.value })?.key

*// Date range*

let sortedByDate = entries.sorted { $0.date < $1.date }

let dateRange: DateInterval?

if let first = sortedByDate.first, let last = sortedByDate.last {

dateRange = DateInterval(start: first.date, end: last.date)

} else {

dateRange = nil

}

let stats = PurchaseStatistics(

totalAmount: totalAmount,

entryCount: entryCount,

averageAmount: averageAmount,

categoryBreakdown: categoryBreakdown,

largestPurchase: largestPurchase,

smallestPurchase: smallestPurchase,

mostFrequentCategory: mostFrequentCategory,

dateRange: dateRange

)

*// Cache the result*

statisticsCache = (entries: entries, stats: stats)

return stats

}

## CRUD Operations with Error Recovery:

### Purchase Management:

swift

*/// Add a new purchase*

public func addPurchase(

amount: Double,

category: String,

date: Date,

note: String? = nil

) async throws {

await performOperation(.adding) {

let entry = try BudgetEntry(

amount: amount,

category: category,

date: date,

note: note

)

try await budgetManager.addEntry(entry)

await loadPurchasesData()

}

}

*/// Update an existing purchase*

public func updatePurchase(\_ entry: BudgetEntry) async throws {

await performOperation(.updating(entry)) {

try await budgetManager.updateEntry(entry)

await loadPurchasesData()

}

}

*/// Delete a purchase*

public func deletePurchase(\_ entry: BudgetEntry) async throws {

await performOperation(.deleting(entry)) {

try await budgetManager.deleteEntry(entry)

*// Update local state immediately for better UX*

allEntries.removeAll { $0.id == entry.id }

await applyFiltersAndSort()

}

}

## Error Handling and Recovery:

### Comprehensive Error Management:

swift

private func performOperation<T>(\_ operationType: OperationType, operation: @escaping () async throws -> T) async {

let startTime = Date()

currentOperation = operationType

*// Update loading state*

switch operationType {case .loading:

loadingState = .loading

case .refreshing:

loadingState = .refreshing

default:

break

}

clearError()

do {

let \_ = try await operation()

*// Update state on success*

if case .loading = operationType {

loadingState = .loaded

lastRefreshDate = Date()

retryCount = 0

} else if case .refreshing = operationType {

loadingState = .loaded

lastRefreshDate = Date()

retryCount = 0

}

currentOperation = nil

recordMetric(String(describing: operationType), duration: Date().timeIntervalSince(startTime))

} catch {

await handleError(AppError.from(error))

currentOperation = nil

}

}

private func handleError(\_ error: AppError) async {

hasError = true

currentError = error

if case .loading = loadingState {

loadingState = .error(error)

} else if case .refreshing = loadingState {

loadingState = .error(error)

}

*// Report to global error handler*

errorHandler.handle(error, context: "Purchases view model")

print("❌ PurchasesViewModel: Error - \(error.errorDescription ?? "Unknown error")")

}

## Integration Points:

### Manager Coordination:

* **BudgetManager:** Primary data source with real-time publisher updates
* **ErrorHandler:** Centralized error management with context tracking
* **SettingsManager:** User preferences and configuration

### Data Flow Architecture:

swift

*// Reactive data flow*

budgetManager.objectWillChange

.receive(on: DispatchQueue.main)

.sink { [weak self] in

Task { [weak self] in

await self?.handleBudgetManagerUpdate()

}

}

.store(in: &cancellables)

private func handleBudgetManagerUpdate() async {

*// Only reload if we're not currently loading*

guard !loadingState.isLoading else { return }

await loadPurchasesData()

}

## Testing Support:

### Comprehensive Testing Interface:

swift

#if DEBUG

extension PurchasesViewModel {

*/// Create test view model*

static func createTestViewModel() -> PurchasesViewModel {

return PurchasesViewModel(

budgetManager: BudgetManager.shared,

errorHandler: ErrorHandler.shared,

settingsManager: SettingsManager.shared

)

}

*/// Load test data*

func loadTestData() async {

let testEntries = [

try! BudgetEntry(amount: 45.67, category: "Groceries", date: Date(), note: "Weekly shopping"),

try! BudgetEntry(amount: 12.50, category: "Transportation", date: Date().addingTimeInterval(-86400), note: "Bus fare"),

try! BudgetEntry(amount: 89.99, category: "Entertainment", date: Date().addingTimeInterval(-172800), note: "Movie tickets"),

try! BudgetEntry(amount: 25.00, category: "Dining", date: Date().addingTimeInterval(-259200), note: "Lunch"),

try! BudgetEntry(amount: 150.00, category: "Utilities", date: Date().addingTimeInterval(-345600), note: "Electric bill")

]

allEntries = testEntries

availableCategories = ["Groceries", "Transportation", "Entertainment", "Dining", "Utilities"]

await applyFiltersAndSort()

loadingState = .loaded

lastRefreshDate = Date()

print("✅ PurchasesViewModel: Loaded test data")

}

*/// Get internal state for testing*

func getInternalStateForTesting() -> (

entryCount: Int,

filteredCount: Int,

categoryCount: Int,

hasError: Bool,

isLoading: Bool,

retryCount: Int

) {

return (

entryCount: allEntries.count,

filteredCount: filteredEntries.count,

categoryCount: availableCategories.count,

hasError: hasError,

isLoading: loadingState.isLoading,

retryCount: retryCount

)

}

*/// Get performance metrics for testing*

func getPerformanceMetricsForTesting() -> [String: TimeInterval] {

return metricsQueue.sync {

return operationMetrics

}

}

}

#endif

This PurchasesViewModel represents **enterprise-level state management** with sophisticated filtering, comprehensive analytics, performance monitoring, and robust error handling that serves as a model for ViewModels throughout the application.

# Section 41: Features/Purchases/Views/PurchaseEntryView.swift

**File Type:** Purchase Creation Interface  
**Purpose:** Comprehensive purchase entry with real-time validation, category management, and sophisticated error handling

## Architecture Overview:

swift

struct PurchaseEntryView: View {

*// MARK: - Environment*

@EnvironmentObject private var budgetManager: BudgetManager

@EnvironmentObject private var themeManager: ThemeManager

@EnvironmentObject private var settingsManager: SettingsManager

@EnvironmentObject private var errorHandler: ErrorHandler

@Environment(\.dismiss) private var dismiss

*// MARK: - State*

@State private var amount = ""

@State private var selectedCategory = ""

@State private var selectedDate = Date()

@State private var note = ""

@State private var isSubmitting = false

@State private var hasUnsavedChanges = false

@State private var showingDiscardAlert = false

*// MARK: - Error State*

@State private var validationErrors: [ValidationError] = []

@State private var submissionError: AppError?

@State private var showingErrorDetails = false

@State private var retryCount = 0

private let maxRetries = 3

}

## Real-Time Validation System:

### Validation Error Structure:

swift

private struct ValidationError: Identifiable {

let id = UUID()

let field: String

let message: String

}

private func validateForm() {

validationErrors.removeAll()

validateAmount()

validateCategory()

validateNote()

}

private func validateAmount() {

validationErrors.removeAll { $0.field == "amount" }

guard !amount.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty else {

validationErrors.append(ValidationError(field: "amount", message: "Amount is required"))

return

}

guard let value = Double(amount) else {

validationErrors.append(ValidationError(field: "amount", message: "Please enter a valid amount"))

return

}

guard value > 0 else {

validationErrors.append(ValidationError(field: "amount", message: "Amount must be greater than zero"))

return

}

guard value <= AppConstants.Validation.maximumTransactionAmount else {

validationErrors.append(ValidationError(field: "amount", message: "Amount exceeds maximum allowed (\(AppConstants.Validation.maximumTransactionAmount.asCurrency))"))

return

}

}

## Advanced Input Formatting:

### Smart Amount Input Processing:

swift

private func formatAmountInput() {

*// Remove any non-numeric characters except decimal point*

let filtered = amount.filter { "0123456789.".contains($0) }

*// Ensure only one decimal point*

let components = filtered.components(separatedBy: ".")

if components.count > 2 {

amount = components[0] + "." + components[1...].joined()

} else {

amount = filtered

}

*// Limit decimal places to 2*

if let decimalIndex = amount.firstIndex(of: ".") {

let afterDecimal = amount[amount.index(after: decimalIndex)...]

if afterDecimal.count > 2 {

let validDecimal = String(afterDecimal.prefix(2))

amount = String(amount[...decimalIndex]) + validDecimal

}

}

*// Limit total length*

if amount.count > 10 {

amount = String(amount.prefix(10))

}

}

## Comprehensive UI Implementation:

### Amount Section with Real-Time Formatting:

swift

private var amountSection: some View {

VStack(alignment: .leading, spacing: 8) {

HStack {

Text("Amount")

.font(.headline)

.foregroundColor(.primary)

Spacer()

if !formattedAmount.isEmpty {

Text(formattedAmount)

.font(.subheadline)

.foregroundColor(themeManager.primaryColor)

.fontWeight(.semibold)

}

}

TextField("0.00", text: $amount)

.keyboardType(.decimalPad)

.textFieldStyle(RoundedBorderTextFieldStyle())

.focused($isAmountFocused)

.font(.title2)

.multilineTextAlignment(.center)

.onReceive(Just(amount)) { \_ in

formatAmountInput()

}

if let error = validationErrors.first(where: { $0.field == "amount" }) {

Text(error.message)

.font(.caption)

.foregroundColor(.red)

}

}

.padding()

.background(Color(.secondarySystemGroupedBackground))

.cornerRadius(12)

}

### Dynamic Category Management:

swift

private var categorySection: some View {

VStack(alignment: .leading, spacing: 8) {

HStack {

Text("Category")

.font(.headline)

.foregroundColor(.primary)

Spacer()

if isLoadingCategories {

ProgressView()

.scaleEffect(0.8)

}

Button("New") {

showingNewCategoryAlert = true

}

.font(.subheadline)

.foregroundColor(themeManager.primaryColor)

}

if categoryError != nil {

Text("Failed to load categories")

.font(.subheadline)

.foregroundColor(.red)

} else if availableCategories.isEmpty {

Text("No categories available")

.font(.subheadline)

.foregroundColor(.secondary)

} else {

Picker("Category", selection: $selectedCategory) {

ForEach(availableCategories, id: \.self) { category in

HStack {

Image(systemName: categoryIcon(for: category))

.foregroundColor(themeManager.colorForCategory(category))

Text(category)

}

.tag(category)

}

}

.pickerStyle(.menu)

.tint(themeManager.primaryColor)

}

if let error = validationErrors.first(where: { $0.field == "category" }) {

Text(error.message)

.font(.caption)

.foregroundColor(.red)

}

}

.padding()

.background(Color(.secondarySystemGroupedBackground))

.cornerRadius(12)

}

## Advanced Error Handling and Recovery:

### Submission with Retry Logic:

swift

private func submitPurchase() async {

*// Validate form first*

validateForm()

guard isFormValid else {

if settingsManager.enableHapticFeedback {

errorFeedback.notificationOccurred(.error)

}

return

}

await MainActor.run {

isSubmitting = true

clearSubmissionError()

}

do {

guard let amountValue = Double(amount) else {

throw AppError.validation(message: "Invalid amount format")

}

let trimmedNote = note.trimmingCharacters(in: .whitespacesAndNewlines)

let finalNote = trimmedNote.isEmpty ? nil : trimmedNote

let entry = try BudgetEntry(

amount: amountValue,

category: selectedCategory,

date: selectedDate,

note: finalNote

)

try await budgetManager.addEntry(entry)

await MainActor.run {

*// Success feedback*

if settingsManager.enableHapticFeedback {

successFeedback.notificationOccurred(.success)

}

*// Reset retry count*

retryCount = 0

*// Dismiss view*

dismiss()

}

} catch {

await handleSubmissionError(AppError.from(error))

}

}

private func handleSubmissionError(\_ error: AppError) async {

await MainActor.run {

isSubmitting = false

submissionError = error

showingErrorDetails = true

retryCount += 1

if settingsManager.enableHapticFeedback {

errorFeedback.notificationOccurred(.error)

}

*// Also report to global error handler*

errorHandler.handle(error, context: "Adding purchase")

}

}

## Category Management Integration:

### New Category Creation:

swift

private func addNewCategory() {

let trimmedName = newCategoryName.trimmingCharacters(in: .whitespacesAndNewlines)

guard !trimmedName.isEmpty else { return }

guard trimmedName.count <= AppConstants.Validation.maxCategoryNameLength else { return }

guard !availableCategories.contains(trimmedName) else { return }

availableCategories.append(trimmedName)

availableCategories.sort()

selectedCategory = trimmedName

newCategoryName = ""

*// Mark as having unsaved changes*

updateUnsavedChangesState()

if settingsManager.enableHapticFeedback {

impactFeedback.impactOccurred()

}

}

## Integration Points:

### Environment Object Dependencies:

* **BudgetManager:** Entry creation and category management
* **ThemeManager:** Consistent styling and color coordination
* **SettingsManager:** User preferences and haptic feedback
* **ErrorHandler:** Centralized error management

### UI Flow Integration:

* **Validation:** Real-time field validation with debouncing
* **Categories:** Dynamic category loading with creation flow
* **Error Recovery:** Comprehensive retry mechanisms
* **User Feedback:** Haptic feedback and visual confirmation

This PurchaseEntryView demonstrates **comprehensive form handling** with enterprise-level validation, error recovery, and user experience optimization patterns used throughout the application.

# Section 42: Features/Purchases/Views/UpdatePurchaseView.swift

**File Type:** Purchase Modification Interface  
**Purpose:** Comprehensive purchase editing with change detection, validation, and sophisticated deletion workflows

## Architecture Overview:

swift

struct UpdatePurchaseView: View {

*// MARK: - Properties*

let entry: BudgetEntry

*// MARK: - Environment*

@EnvironmentObject private var budgetManager: BudgetManager

@EnvironmentObject private var themeManager: ThemeManager

@EnvironmentObject private var settingsManager: SettingsManager

@EnvironmentObject private var errorHandler: ErrorHandler

@Environment(\.dismiss) private var dismiss

*// MARK: - State*

@State private var amount: String

@State private var selectedCategory: String

@State private var selectedDate: Date

@State private var note: String

@State private var hasUnsavedChanges = false

*// MARK: - Original Values (for change detection)*

private let originalAmount: String

private let originalCategory: String

private let originalDate: Date

private let originalNote: String

*// MARK: - Initialization*

init(entry: BudgetEntry) {

self.entry = entry

self.originalAmount = String(entry.amount)

self.originalCategory = entry.category

self.originalDate = entry.date

self.originalNote = entry.note ?? ""

*// Initialize state*

self.\_amount = State(initialValue: String(entry.amount))

self.\_selectedCategory = State(initialValue: entry.category)

self.\_selectedDate = State(initialValue: entry.date)

self.\_note = State(initialValue: entry.note ?? "")

}

}

## Advanced Change Detection System:

### Real-Time Change Monitoring:

swift

private func updateUnsavedChangesState() {

let currentNote = note.trimmingCharacters(in: .whitespacesAndNewlines)

let originalNoteForComparison = originalNote.trimmingCharacters(in: .whitespacesAndNewlines)

hasUnsavedChanges = (

amount != originalAmount ||

selectedCategory != originalCategory ||

!Calendar.current.isDate(selectedDate, inSameDayAs: originalDate) ||

currentNote != originalNoteForComparison

)

}

*// onChange modifiers for real-time detection*

.onChange(of: amount) { \_, \_ in

validateAmount()

updateUnsavedChangesState()

}

.onChange(of: selectedCategory) { \_, \_ in

validateCategory()

updateUnsavedChangesState()

}

.onChange(of: selectedDate) { \_, \_ in

updateUnsavedChangesState()

}

.onChange(of: note) { \_, \_ in

validateNote()

updateUnsavedChangesState()

}

## Original Purchase Display:

### Visual Comparison Interface:

swift

private var originalPurchaseInfo: some View {

VStack(spacing: 12) {

HStack {

Text("Original Purchase")

.font(.headline)

.foregroundColor(.secondary)

Spacer()

}

HStack(spacing: 16) {

*// Category icon*

Circle()

.fill(themeManager.colorForCategory(entry.category))

.frame(width: 50, height: 50)

.overlay(

Image(systemName: categoryIcon(for: entry.category))

.foregroundColor(.white)

.font(.system(size: 20, weight: .semibold))

)

VStack(alignment: .leading, spacing: 4) {

Text(entry.category)

.font(.headline)

.foregroundColor(.primary)

if let originalNote = entry.note, !originalNote.isEmpty {

Text(originalNote)

.font(.subheadline)

.foregroundColor(.secondary)

.lineLimit(2)

}

Text(entry.formattedDate)

.font(.caption)

.foregroundColor(.tertiary)

}

Spacer()

VStack(alignment: .trailing, spacing: 4) {

Text(entry.formattedAmount)

.font(.title2)

.fontWeight(.bold)

.foregroundColor(themeManager.primaryColor)

Text("Original")

.font(.caption)

.foregroundColor(.secondary)

}

}

}

.padding()

.background(Color(.secondarySystemGroupedBackground))

.cornerRadius(12)

.padding(.horizontal)

}

## Enhanced Toolbar with Multiple Actions:

### Context-Sensitive Toolbar:

swift

.toolbar {

ToolbarItem(placement: .navigationBarLeading) {

Button("Cancel") {

handleCancelAction()

}

.disabled(isSubmitting)

}

ToolbarItemGroup(placement: .navigationBarTrailing) {

Button {

showingDeleteAlert = true

} label: {

Image(systemName: "trash")

.foregroundColor(.red)

}

.disabled(!canDelete)

Button("Save") {

Task {

await updatePurchase()

}

}

.fontWeight(.semibold)

.disabled(!canSubmit)

}

}

## Advanced CRUD Operations:

### Update Operation with Validation:

swift

private func updatePurchase() async {

*// Validate form first*

validateForm()

guard isFormValid else {

if settingsManager.enableHapticFeedback {

errorFeedback.notificationOccurred(.error)

}

return

}

await MainActor.run {

isSubmitting = true

clearSubmissionError()

}

do {

guard let amountValue = Double(amount) else {

throw AppError.validation(message: "Invalid amount format")

}

let trimmedNote = note.trimmingCharacters(in: .whitespacesAndNewlines)

let finalNote = trimmedNote.isEmpty ? nil : trimmedNote

let updatedEntry = try BudgetEntry(

id: entry.id,

amount: amountValue,

category: selectedCategory,

date: selectedDate,

note: finalNote

)

try await budgetManager.updateEntry(updatedEntry)

await MainActor.run {

*// Success feedback*

if settingsManager.enableHapticFeedback {

successFeedback.notificationOccurred(.success)

}

*// Reset retry count*

retryCount = 0

*// Dismiss view*

dismiss()

}

} catch {

await handleSubmissionError(AppError.from(error))

}

}

### Delete Operation with Confirmation:

swift

private func deletePurchase() async {

await MainActor.run {

isSubmitting = true

clearSubmissionError()

}

do {

try await budgetManager.deleteEntry(entry)

await MainActor.run {

*// Success feedback*

if settingsManager.enableHapticFeedback {

successFeedback.notificationOccurred(.success)

}

*// Reset retry count*

retryCount = 0

*// Dismiss view*

dismiss()

}

} catch {

await handleSubmissionError(AppError.from(error))

}

}

## Sophisticated Alert System:

### Multi-Level Confirmation Dialogs:

swift

.alert("Discard Changes?", isPresented: $showingDiscardAlert) {

Button("Discard", role: .destructive) {

dismiss()

}

Button("Keep Editing", role: .cancel) { }

} message: {

Text("You have unsaved changes. Are you sure you want to discard them?")

}

.alert("Delete Purchase", isPresented: $showingDeleteAlert) {

Button("Delete", role: .destructive) {

Task {

await deletePurchase()

}

}

Button("Cancel", role: .cancel) { }

} message: {

Text("Are you sure you want to delete this purchase? This action cannot be undone.")

}

.alert("Purchase Error", isPresented: $showingErrorDetails, presenting: submissionError) { error in

if error.isRetryable && retryCount < maxRetries {

Button("Retry") {

Task {

await updatePurchase()

}

}

}

Button("OK", role: .cancel) {

clearSubmissionError()

}

} message: { error in

VStack(alignment: .leading, spacing: 4) {

Text(error.errorDescription ?? "Failed to update purchase")

if let suggestion = error.recoverySuggestion {

Text(suggestion)

.font(.caption)

}

}

}

## Category Icon System:

### Comprehensive Category Mapping:

swift

private func categoryIcon(for category: String) -> String {

switch category.lowercased() {

case "groceries", "food": return "cart.fill"

case "transportation", "transport": return "car.fill"

case "entertainment": return "gamecontroller.fill"

case "utilities": return "bolt.fill"

case "healthcare", "medical": return "cross.fill"

case "shopping": return "bag.fill"

case "dining", "restaurant": return "fork.knife"

case "education": return "book.fill"

case "savings": return "banknote.fill"

case "housing": return "house.fill"

case "insurance": return "shield.fill"

case "personal care": return "figure.walk"

default: return "creditcard.fill"

}

}

## Integration Points:

### Computed Properties for UI State:

swift

private var canSubmit: Bool {

return isFormValid && hasUnsavedChanges && !isSubmitting

}

private var canDelete: Bool {

return !isSubmitting

}

private var shouldShowDiscardAlert: Bool {

return hasUnsavedChanges

}

### Error Recovery Integration:

* **Retry Logic:** Configurable retry attempts with exponential backoff
* **Validation Display:** Real-time validation with field-specific errors
* **Context Preservation:** Maintains user input during error recovery
* **Graceful Degradation:** Handles network failures and data conflicts

This UpdatePurchaseView demonstrates **sophisticated editing workflows** with comprehensive change detection, multi-level confirmation systems, and enterprise-level error handling patterns used throughout the application.

# Section 43: Features/Settings/Models/SettingsStorage.swift

**File Type:** Type-Safe Settings Storage System  
**Purpose:** Advanced property wrapper for robust settings persistence with validation, transformation, and reactive programming

## Architecture Overview:

swift

@propertyWrapper

public struct SettingsStorage<T: Codable> where T: Codable {

*// MARK: - Properties*

private let key: String

private let defaultValue: T

private let storage: UserDefaults

private let validator: ((T) -> Bool)?

private let transformer: ((T) -> T)?

*// Publisher for value changes*

public let publisher = PassthroughSubject<T, Never>()

*// MARK: - Initialization*

public init(

wrappedValue defaultValue: T,

key: String,

storage: UserDefaults = .standard,

validator: ((T) -> Bool)? = nil,

transformer: ((T) -> T)? = nil

) {

self.key = key

self.defaultValue = defaultValue

self.storage = storage

self.validator = validator

self.transformer = transformer

}

}

## Advanced Error Management:

### Comprehensive Error Types:

swift

public enum StorageError: LocalizedError {

case encodingFailed

case decodingFailed

case invalidValue

case storageAccessFailed

public var errorDescription: String? {

switch self {

case .encodingFailed: return "Failed to encode settings value"

case .decodingFailed: return "Failed to decode settings value"

case .invalidValue: return "Invalid settings value"

case .storageAccessFailed: return "Failed to access storage"

}

}

}

## Property Wrapper Implementation:

### Safe Value Access with Error Recovery:

swift

public var wrappedValue: T {

get {

do {

return try getValue()

} catch {

print("Failed to get settings value: \(error.localizedDescription)")

return defaultValue

}

}

set {

do {

try setValue(newValue)

publisher.send(newValue)

} catch {

print("Failed to set settings value: \(error.localizedDescription)")

}

}

}

private func getValue() throws -> T {

guard let data = storage.data(forKey: key) else {

return defaultValue

}

do {

let decoder = JSONDecoder()

let value = try decoder.decode(T.self, from: data)

if let transformer = transformer {

return transformer(value)

}

return value

} catch {

throw StorageError.decodingFailed

}

}

private func setValue(\_ newValue: T) throws {

if let validator = validator, !validator(newValue) {

throw StorageError.invalidValue

}

let valueToStore = transformer?(newValue) ?? newValue

do {

let encoder = JSONEncoder()

let data = try encoder.encode(valueToStore)

storage.set(data, forKey: key)

} catch {

throw StorageError.encodingFailed

}

}

## Reactive Programming Integration:

### Publisher Projection:

swift

public var projectedValue: SettingsStorageProjection<T> {

SettingsStorageProjection<T>(storage: self)

}

public struct SettingsStorageProjection<T: Encodable & Decodable> {

private let storage: SettingsStorage<T>

init(storage: SettingsStorage<T>) {

self.storage = storage

}

*/// Publisher for value changes*

public var publisher: AnyPublisher<T, Never> {

storage.publisher.eraseToAnyPublisher()

}

}

## UserDefaults Extensions:

### Comprehensive Key Management:

swift

extension UserDefaults {

*/// Settings keys namespace*

public enum Keys {

public static let userName = "userName"

public static let defaultCurrency = "defaultCurrency"

public static let isDarkMode = "isDarkMode"

public static let primaryColor = "primaryColor"

public static let notificationsAllowed = "notificationsAllowed"

public static let purchaseNotificationsEnabled = "purchaseNotificationsEnabled"

public static let purchaseNotificationFrequency = "purchaseNotificationFrequency"

public static let budgetTotalNotificationsEnabled = "budgetTotalNotificationsEnabled"

public static let budgetTotalNotificationFrequency = "budgetTotalNotificationFrequency"

public static let isFirstLaunch = "isFirstLaunch"

*// Constants for key prefixes*

public static let prefix = "com.brandonsbudget.settings."

*/// Generate prefixed key*

public static func prefixed(\_ key: String) -> String {

prefix + key

}

}

*/// Check if key exists*

func contains(key: String) -> Bool {

object(forKey: key) != nil

}

*/// Remove all app settings*

func removeAllSettings() {

let prefix = Keys.prefix

let allKeys = dictionaryRepresentation().keys

let settingsKeys = allKeys.filter { $0.hasPrefix(prefix) }

settingsKeys.forEach { removeObject(forKey: $0) }

}

}

## Testing Support:

### Comprehensive Testing Infrastructure:

swift

#if DEBUG

extension UserDefaults {

*/// Create test storage*

static func createTestStorage(suiteName: String) -> UserDefaults? {

let defaults = UserDefaults(suiteName: suiteName)

defaults?.removeAllSettings()

return defaults

}

}

extension SettingsStorage {

*/// Create test storage*

static func createTestStorage(

defaultValue: T,

key: String

) -> SettingsStorage<T> {

let testDefaults = UserDefaults.createTestStorage(suiteName: "test\_settings")!

return SettingsStorage(

wrappedValue: defaultValue,

key: key,

storage: testDefaults

)

}

}

#endif

## Usage Patterns:

### Type-Safe Settings with Validation:

swift

*// Example usage in SettingsManager*

@SettingsStorage(

key: "userName",

validator: { !$0.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty }

)

var userName: String = ""

@SettingsStorage(

key: "defaultCurrency",

validator: { supportedCurrencies.contains($0) }

)

var defaultCurrency: String = "USD"

@SettingsStorage(

key: "budgetLimit",

transformer: { max(0, min($0, 1000000)) } *// Clamp to valid range*

) var budgetLimit: Double = 5000.0

// Reactive programming with publishers

$userName.publisher

.debounce(for: .milliseconds(500), scheduler: DispatchQueue.main)

.sink { newName in

// React to username changes

updateUserProfile(name: newName)

}

.store(in: &cancellables)

**Advanced Validation System:**

**Custom Validators:**

swift

// Email validation

@SettingsStorage(

key: "userEmail",

validator: { email in

let emailRegex = #"^[A-Z0-9a-z.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$"#

return email.range(of: emailRegex, options: .regularExpression) != nil

}

)

var userEmail: String = ""

// Currency validation

@SettingsStorage(

key: "defaultCurrency",

validator: { currency in

return ["USD", "EUR", "GBP", "JPY", "CAD", "AUD", "CHF", "CNY", "SEK", "NOK", "DKK"].contains(currency)

}

)

var defaultCurrency: String = "USD"

// Amount range validation

@SettingsStorage(

key: "monthlyBudget",

validator: { amount in

return amount >= 0 && amount <= 100000

},

transformer: { amount in

return max(0, min(amount, 100000)) // Clamp to valid range

}

)

var monthlyBudget: Double = 3000.0

## Migration and Transformation:

### Version Migration Support:

swift

*// Version-aware settings with migration*

@SettingsStorage(

key: "appSettings\_v2",

transformer: { settings in

*// Migrate from v1 to v2 format*

var migratedSettings = settings

*// Example: Convert old boolean to new enum*

if let oldValue = settings["useMetricUnits"] as? Bool {

migratedSettings["unitSystem"] = oldValue ? "metric" : "imperial"

migratedSettings.removeValue(forKey: "useMetricUnits")

}

return migratedSettings

}

)

var appSettings: [String: Any] = [:]

*// Complex object transformation*

struct UserPreferences: Codable {

var theme: String

var notifications: Bool

var language: String

*// Migration support*

init(from decoder: Decoder) throws {

let container = try decoder.container(keyedBy: CodingKeys.self)

*// Handle old theme values*

if let oldTheme = try? container.decode(String.self, forKey: .theme) {

switch oldTheme {

case "light", "dark": theme = oldTheme

default: theme = "system" *// New default*

}

} else {

theme = "system"

}

notifications = try container.decodeIfPresent(Bool.self, forKey: .notifications) ?? true

language = try container.decodeIfPresent(String.self, forKey: .language) ?? "en"

}

}

## Performance Optimization:

### Efficient Storage Operations:

swift

*// Batched updates for related settings*

extension SettingsStorage {

*/// Perform multiple updates atomically*

static func batchUpdate<T1: Codable, T2: Codable>(

\_ storage1: SettingsStorage<T1>, value1: T1,

\_ storage2: SettingsStorage<T2>, value2: T2

) throws {

*// Use UserDefaults batch update*

let userDefaults = storage1.storage

*// Validate all values first*

if let validator1 = storage1.validator, !validator1(value1) {

throw StorageError.invalidValue

}

if let validator2 = storage2.validator, !validator2(value2) {

throw StorageError.invalidValue

}

*// Encode all values*

let encoder = JSONEncoder()

let data1 = try encoder.encode(storage1.transformer?(value1) ?? value1)

let data2 = try encoder.encode(storage2.transformer?(value2) ?? value2)

*// Set all values atomically*

userDefaults.set(data1, forKey: storage1.key)

userDefaults.set(data2, forKey: storage2.key)

*// Notify all publishers*

storage1.publisher.send(value1)

storage2.publisher.send(value2)

}

}

## Integration with SwiftUI:

### Binding Support:

swift

extension SettingsStorage {

*/// Create a SwiftUI binding*

var binding: Binding<T> {

Binding(

get: { self.wrappedValue },

set: { self.wrappedValue = $0 }

)

}

}

*// Usage in SwiftUI views*

struct SettingsView: View {

@SettingsStorage(key: "userName") var userName: String = ""

@SettingsStorage(key: "enableNotifications") var enableNotifications: Bool = true

var body: some View {

Form {

TextField("Name", text: $userName.binding)

Toggle("Enable Notifications", isOn: $enableNotifications.binding)

}

}

}

## Error Recovery and Resilience:

### Graceful Degradation:

swift

extension SettingsStorage {

*/// Attempt recovery from corrupted data*

mutating func recover() {

do {

*// Try to read the value*

let \_ = try getValue()

} catch {

*// If reading fails, reset to default and clear corrupted data*

storage.removeObject(forKey: key)

publisher.send(defaultValue)

print("⚠️ SettingsStorage: Recovered from corrupted data for key '\(key)'")

}

}

*/// Validate current stored value*

var isValid: Bool {

do {

let value = try getValue()

return validator?(value) ?? true

} catch {

return false

}

}

}

## Debug and Monitoring:

### Development Tools:

swift

#if DEBUG

extension SettingsStorage {

*/// Get debug information*

var debugInfo: [String: Any] {

return [

"key": key,

"hasValidator": validator != nil,

"hasTransformer": transformer != nil,

"isValid": isValid,

"rawDataSize": storage.data(forKey: key)?.count ?? 0,

"defaultValue": String(describing: defaultValue)

]

}

*/// Force reset to default*

mutating func resetToDefault() {

storage.removeObject(forKey: key)

publisher.send(defaultValue)

}

}

*// Global settings monitoring*

struct SettingsMonitor {

static func logAllSettings() {

let allKeys = UserDefaults.standard.dictionaryRepresentation().keys

let settingsKeys = allKeys.filter { $0.hasPrefix(UserDefaults.Keys.prefix) }

print("📊 Settings Monitor - Active Settings:")

for key in settingsKeys.sorted() {

let value = UserDefaults.standard.object(forKey: key)

print(" \(key): \(String(describing: value))")

}

}

}

#endif

## Integration Points:

### Manager Integration:

* **SettingsManager:** Primary consumer for all app settings
* **ThemeManager:** Theme preferences and color coordination
* **NotificationManager:** Notification preferences and scheduling
* **Performance:** Automatic validation and error recovery

### Reactive Coordination:

* **Publisher Integration:** Combine-based reactive updates
* **Change Propagation:** Automatic UI updates via ObservableObject
* **Validation Feedback:** Real-time validation with error messaging
* **Migration Support:** Version-aware settings transformation

This SettingsStorage system provides **enterprise-level settings management** with type safety, validation, reactive programming, and comprehensive error handling that serves as the foundation for all app configuration throughout the application.

**44. Features/Settings/Views/SettingsView.swift**

**File Type:** Main Settings Interface  
**Purpose:** Comprehensive settings management with data operations, backup, and system integration

**Architecture:**

swift

struct SettingsView: View {

@EnvironmentObject private var budgetManager: BudgetManager

@EnvironmentObject private var themeManager: ThemeManager

@EnvironmentObject private var settingsManager: SettingsManager

@EnvironmentObject private var errorHandler: ErrorHandler

@EnvironmentObject private var notificationManager: NotificationManager

@State private var showingExportOptions = false

@State private var showingImportOptions = false

@State private var loadingStates: Set<String> = []

}

**Settings Categories:**

1. **User Section:**
   * Name input with real-time validation
   * Currency picker with supported currencies
   * Input validation and error handling
2. **Notification Section:**
   * Master notification toggle with system integration
   * Status display with system permission awareness
   * Navigation to detailed notification settings
3. **Data Management Section:**
   * Export data with options sheet
   * Import data with validation and preview
   * Backup & sync with status display
   * Data health monitoring
   * Reset app data with confirmation
4. **Appearance Section:**
   * Theme color picker with live preview
   * Dark mode toggle
   * Display preferences (decimal places, haptic feedback)
5. **Privacy & Security Section:**
   * Privacy mode toggle
   * Biometric authentication (when available)
   * Data export format preferences
6. **Advanced Section:**
   * Advanced features toggle
   * Debug tools and diagnostics
   * Performance optimization
7. **About Section:**
   * App version and build information
   * Privacy policy, terms, support links
   * App information and credits

**Data Operations:**

swift

private func performExport() async

private func shareExportFile(\_ url: URL)

private func handleCategoryMapping(\_ mappings: [String: String]) async

private func performReset() async

private func performDiagnostics() async

**Advanced Features:**

1. **Loading States Management:**
   * Per-operation loading indicators
   * Non-blocking UI with progress feedback
   * Error recovery with retry mechanisms
2. **Import/Export Coordination:**
   * Category mapping integration
   * Progress tracking for large operations
   * Validation and preview before final operations
3. **System Integration:**
   * Notification permission handling
   * Biometric authentication detection
   * System settings integration

**Error Handling:**

* **Inline Errors:** InlineErrorView for recoverable errors
* **Alert Dialogs:** System alerts for critical errors
* **Loading Overlays:** Progress indication during operations
* **Validation Display:** Real-time validation feedback

**Integration Points:**

* Central hub for all app configuration
* Coordinates with all manager classes
* Handles complex data operations (import/export/backup)
* Provides system integration (notifications, biometrics)

**45. Features/Settings/Views/ExportOptionsView.swift**

**File Type:** Advanced Export Configuration Interface  
**Purpose:** Comprehensive export system with validation, preview, and multiple format support

**Architecture:**

swift

struct ExportOptionsView: View {

@Binding var exportTimePeriod: TimePeriod

let onExport: () -> Void

@State private var selectedExportType: CSVExport.ExportType = .budgetEntries

@State private var includeCurrency = true

@State private var includeHeaders = true

@State private var decimalPlaces = 2

@State private var previewData: ExportPreview?

}

**Export Configuration:**

swift

private struct ExportPreview {

let recordCount: Int

let estimatedSize: String

let dateRange: String

let categories: [String]

let totalAmount: Double

let sampleData: String

}

**UI Sections:**

1. **Export Type Selection:**
   * Budget Entries, Monthly Budgets, Combined Data
   * Visual cards with descriptions and icons
   * Interactive selection with animation
2. **Time Period Configuration:**
   * All TimePeriod options with descriptions
   * Custom range selection with date pickers
   * Duration calculation and validation
3. **Format Options:**
   * Currency symbol inclusion
   * Column headers toggle
   * Decimal places configuration
   * Encoding and delimiter options
4. **Live Preview:**
   * Real-time data analysis
   * Record count and file size estimation
   * Sample data preview
   * Warning indicators for large exports
5. **Export Details:**
   * File format specifications
   * Compatibility information
   * Field descriptions for each export type

**Advanced Features:**

1. **Validation System:**
   * Precondition checking (data availability, disk space)
   * Export size estimation
   * Custom date range validation
2. **Preview Generation:**

swift

private func generatePreview() async

private func estimateFileSize(recordCount: Int) -> String

private func generateSampleData(for type: CSVExport.ExportType, from data: [Any]) -> String

1. **Error Handling:**
   * Validation error display
   * Recovery suggestions
   * Graceful fallbacks for edge cases

**Export Process:**

* **Configuration Validation:** Pre-export validation
* **Preview Generation:** Async data analysis
* **Export Execution:** Progress tracking
* **Result Handling:** File sharing and cleanup

**Integration Points:**

* Coordinates with CSVExport service for data processing
* Uses TimePeriod for sophisticated date filtering
* Integrates with BudgetManager for data access
* Provides comprehensive error handling and user feedback

**46. Features/Settings/Views/ImportOptionsView.swift**

**File Type:** Advanced Import System Interface  
**Purpose:** Comprehensive import system with validation, category mapping, and error recovery

**Architecture:**

swift

struct ImportOptionsView: View {

@Binding var showingImportBudgetPicker: Bool

@Binding var showingImportPurchasePicker: Bool

@State private var importConfiguration = CSVImport.ImportConfiguration.default

@State private var recentImports: [RecentImport] = []

@State private var isValidatingFile = false

}

**Import System:**

swift

private enum ImportType {

case budget, purchases, autoDetect

var displayName: String, description: String, icon: String

var expectedFormat: String, sampleData: String

}

private struct RecentImport {

let fileName: String, type: ImportType

let recordCount: Int, date: Date, success: Bool

}

**UI Components:**

1. **Quick Start Section:**
   * Auto-detect, Purchases, Budget options
   * Visual cards with comprehensive descriptions
   * One-tap import initiation
2. **Format Information:**
   * Expandable format specifications
   * Expected headers and sample data
   * Copy-pasteable format examples
3. **Import Options:**
   * Detailed import buttons with status indicators
   * Loading states during validation
   * Error handling with retry mechanisms
4. **Advanced Section:**
   * Configuration options sheet
   * Import guide with step-by-step instructions
   * Template download functionality
5. **Recent Imports:**
   * Import history with success/failure indicators
   * File details and record counts
   * Re-import capabilities

**Advanced Features:**

1. **Template System:**
   * Downloadable CSV templates
   * Sample data generation
   * Share sheet integration
2. **Import Guide:**
   * Step-by-step instructions
   * Pro tips and common issues
   * Visual guidance with examples
3. **Advanced Configuration:**
   * Validation options (strict vs permissive)
   * File size and row limits
   * Date format specifications
   * Encoding and delimiter settings

**Validation and Error Handling:**

swift

private func handleImportError(\_ error: CSVImport.ImportError)

* **File Validation:** Size, format, encoding checks
* **Data Validation:** Header verification, content analysis
* **Error Recovery:** Specific error messages with solutions
* **User Guidance:** Contextual help and suggestions

**Integration Points:**

* Coordinates with CSVImport service for processing
* Integrates with CategoryMappingView for category resolution
* Uses file picker for document selection
* Provides comprehensive error handling and user feedback

**47. Features/Settings/Views/CategoryMappingView.swift**

**File Type:** Advanced Category Resolution Interface  
**Purpose:** Sophisticated category mapping with validation, conflict resolution, and user guidance

**Architecture:**

swift

struct CategoryMappingView: View {

let categories: Set<String>

let importedData: [CSVImport.PurchaseImportData]

let onComplete: ([String: String]) -> Void

@State private var categoryMappings: [String: String] = [:]

@State private var validationErrors: [String: String] = [:]

@State private var showingProgressView = false

@State private var completionPercentage: Double = 0

}

**Mapping System:**

swift

private var categoriesWithCounts: [(category: String, count: Int, totalAmount: Double)]

private var unmappedCategories: [String]

private var mappedCategories: [String]

private var isMappingComplete: Bool

**UI Components:**

1. **Instructions Header:**
   * Dynamic guidance based on available categories
   * Progress indicators (mapped/remaining/errors)
   * Blocking message display for incomplete mappings
2. **Import Summary:**
   * Transaction count and total amount
   * Category breakdown with counts and amounts
   * Visual summary cards with processing status
3. **Unmapped Categories Section:**
   * Categories requiring mapping decisions
   * Map to existing category dropdowns
   * Create new category buttons
   * Validation error display
4. **Mapped Categories Section:**
   * Successfully mapped categories
   * Original → New mapping display
   * Unmap functionality for corrections
5. **Validation Errors Section:**
   * Field-specific error display
   * Retry mechanisms for failed validations
   * Bulk error resolution options

**Category Creation Flow:**

swift

private var createNewCategorySheet: some View {

*// Complete category creation with budget amount*

*// Future month coordination*

*// Validation and confirmation*

}

**Advanced Features:**

1. **Smart Mapping:**
   * Automatic suggestions for similar category names
   * Estimated budget amounts based on import data
   * Future month coordination for new categories
2. **Validation System:**
   * Real-time category name validation
   * Conflict detection and resolution
   * Budget amount validation
3. **Progress Tracking:**
   * Visual progress overlay for category creation
   * Completion percentage calculation
   * Cancellation support for long operations

**Error Handling and Recovery:**

* **Validation Errors:** Field-specific error display with suggestions
* **Category Conflicts:** Detection and resolution workflows
* **Network Errors:** Retry mechanisms for category creation
* **State Recovery:** Proper cleanup on cancellation or errors

**Integration Points:**

* Receives import data from ImportOptionsView
* Coordinates with BudgetManager for category creation
* Uses MoneyCalculatorView for budget amount input
* Provides completion callback with mapping results

**48. Features/Settings/ViewModels/SettingsViewModel.swift**

**File Type:** Advanced Settings Management Logic  
**Purpose:** Comprehensive settings state management with validation, operations, and system integration

**Architecture:**

swift

@MainActor

public final class SettingsViewModel: ObservableObject {

@Published public var operationState: OperationState = .idle

@Published public var exportOperation: ExportOperation = .none

@Published public var importOperation: ImportOperation = .none

@Published public var validationIssues: [ValidationIssue] = []

@Published public var systemHealth: SystemHealth = SystemHealth()

}

**State Management:**

swift

public enum OperationState {

case idle, loading, success(String), error(AppError)

}

public enum ExportOperation {

case none, preparing, exporting, complete(URL), failed(AppError)

var isActive: Bool, progress: Double

}

public enum ImportOperation {

case none, validating, processing, mappingCategories([String])

case importing, complete(ImportResult), failed(AppError)

var isActive: Bool, progress: Double

}

**Validation System:**

swift

public struct ValidationIssue: Identifiable {

public let field: String, message: String

public let severity: AppError.ErrorSeverity

public let suggestion: String?

}

private func validateCurrentSettings()

private func validateSetting(\_ setting: String, value: Any) -> ValidationIssue?

private func validateAllSettings() -> [ValidationIssue]

**System Health Monitoring:**

swift

public struct SystemHealth {

public enum HealthLevel: String, CaseIterable {

case healthy, caution, warning, critical

var color: Color, systemImageName: String

}

public let level: HealthLevel

public let issues: [String], recommendations: [String]

public let lastUpdated: Date

var isHealthy: Bool, needsAttention: Bool, summary: String

}

**Core Operations:**

swift

public func updateUserName() async

public func updateDefaultCurrency() async

public func updateNotificationSettings(...) async

public func performDataExport() async

public func handlePurchaseImport(from url: URL) async

public func completeCategoryMapping(with mappings: [String: String]) async

public func resetAllAppData() async

public func createDataBackup() async

**Advanced Analytics:**

swift

public func getExportStatistics() -> [String: Any]

public func getImportCapabilities() -> [String: Any]

public func getNotificationConfiguration() async -> [String: Any]

public func performSystemCheck() async -> SystemDiagnostic

**Performance Monitoring:**

* **Operation Metrics:** Timing for all settings operations
* **Memory Management:** Proper cleanup and resource management
* **Background Processing:** Heavy operations off main thread
* **Error Recovery:** Comprehensive retry and recovery mechanisms

**Integration Points:**

* Coordinates with all manager classes for comprehensive settings management
* Manages complex import/export workflows
* Provides system health monitoring and diagnostics
* Handles validation and error recovery across all settings operations

**🎨 Utils System (Utils/)**

**Extensions (Utils/Extensions/)**

## Section 49: Utils/Extensions/DateExtensions.swift

**File Type:** Date Utility Extensions  
**Purpose:** Comprehensive date manipulation with thread safety and robust error handling

**Architecture:**

swift

public extension Date {

*// Thread-safe calendar instance*

private static let calendar: Calendar = {

var calendar = Calendar(identifier: .gregorian)

calendar.timeZone = .current

return calendar

}()

}

**Core Calendar Operations:**

swift

*// Boundary calculations*

var startOfDay: Date *// Beginning of day (00:00:00)*

var endOfDay: Date *// End of day (23:59:59)*

var startOfMonth: Date *// First day of month*

var endOfMonth: Date *// Last day of month*

var startOfYear: Date *// January 1st*

var endOfYear: Date *// December 31st*

*// Component access*

var month: Int *// 1-12*

var year: Int *// Full year*

var day: Int *// Day of month*

var weekday: Int *// 1-7 (Sunday = 1)*

**Date Checks and Validation:**

swift

var isToday: Bool *// Is today's date*

var isYesterday: Bool *// Is yesterday's date*

var isTomorrow: Bool *// Is tomorrow's date*

var isInCurrentMonth: Bool *// Same month/year as now*

var isInCurrentYear: Bool *// Same year as now*

var isPast: Bool *// Before current time*

var isFuture: Bool *// After current time*

**Advanced Date Manipulation:**

swift

func adding(days: Int) -> Date *// Add/subtract days*

func adding(months: Int) -> Date *// Add/subtract months*

func adding(years: Int) -> Date *// Add/subtract years*

func daysBetween(\_ date: Date) -> Int *// Days difference*

func monthsBetween(\_ date: Date) -> Int *// Months difference*

func yearsBetween(\_ date: Date) -> Int *// Years difference*

**Formatting Utilities:**

swift

var relativeDescription: String *// "2 days ago"*

var monthYearString: String *// "January 2024"*

func formatted(style: DateFormatter.Style = .medium) -> String

*// Specialized formatters*

static func from(year: Int, month: Int, day: Int) -> Date?

static func fromISO8601(\_ string: String) -> Date?

**Testing Support:**

swift

#if DEBUG

static func testDate(year: Int, month: Int, day: Int) -> Date

#endif

**Integration Points:**

* Used throughout app for consistent date handling
* Thread-safe implementation prevents calendar conflicts
* Provides localized formatting with timezone awareness
* Supports testing with mock date generation

## Section 50: Utils/Extensions/NumberFormatterExtensions.swift

**File Type:** Number Formatting System  
**Purpose:** Thread-safe number formatting with caching and comprehensive currency support

**Architecture:**

swift

public extension NumberFormatter {

*// Thread-safe formatter cache*

private static let formatterCache = NSCache<NSString, NumberFormatter>()

}

**Core Formatters:**

swift

static var currencyFormatter: NumberFormatter *// Thread-safe currency*

static var percentageFormatter: NumberFormatter *// Percentage display*

static var decimalFormatter: NumberFormatter *// Decimal numbers*

*// Advanced formatting methods*

static func formatCurrency(\_ value: Double, locale: Locale = .current) -> String

static func formatPercentage(\_ value: Double) -> String

static func formatDecimal(\_ value: Double) -> String

static func formatWithCurrency(\_ value: Double, currencyCode: String) -> String

static func formatWithSeparators(\_ value: Double) -> String

static func formatCompact(\_ value: Double) -> String *// 1.2K, 1.5M notation*

**Currency System:**

* **Locale Support:** Automatic currency symbol and formatting rules
* **Custom Currency Codes:** Support for specific currency formatting
* **Thread Safety:** Cached formatters prevent thread conflicts
* **Performance:** NSCache automatically manages memory

**Compact Notation:**

swift

*// Examples of compact formatting:*

1234 → "1.2K"

1234567 → "1.2M"

1234567890 → "1.2B"

**Extensions Integration:**

swift

extension Decimal {

var asCurrency: String *// Decimal → Currency string*

}

extension Double {

var asCurrency: String *// Double → Currency string*

}

**Testing Support:**

swift

#if DEBUG

static func clearCache() *// Clear cache for testing*

static var cacheCount: Int *// Cache statistics*

#endif

**Integration Points:**

* Used by all financial display components
* Provides consistent currency formatting across features
* Thread-safe for concurrent usage in background operations
* Supports localization and multiple currency codes

## Section 51: Utils/Extensions/ColorsExtensions.swift

**File Type:** Advanced Color System  
**Purpose:** Comprehensive color management with accessibility, validation, and advanced color theory

**Architecture:**

swift

public struct ColorComponents: Codable, Equatable, Hashable {

public let red: Double *// 0.0-1.0*

public let green: Double *// 0.0-1.0*

public let blue: Double *// 0.0-1.0*

public let opacity: Double *// 0.0-1.0*

*// Validation and safety*

public init(red: Double, green: Double, blue: Double, opacity: Double = 1.0)

public init(from color: Color)

public init?(hex: String) *// Hex string parsing*

public init(r: Int, g: Int, b: Int, opacity: Double = 1.0) *// RGB 0-255*

}

**Color Creation and Validation:**

swift

public extension Color {

init(\_ components: ColorComponents)

init(r: Int, g: Int, b: Int, opacity: Double = 1) throws

init(hex: String) throws *// Comprehensive hex validation*

init(hue: Double, saturation: Double, brightness: Double, opacity: Double = 1.0) throws

}

**Color Properties and Analysis:**

swift

var hexString: String *// #FF0000 format*

var colorComponents: ColorComponents *// RGBA components*

var brightness: Double *// Luminance calculation*

var isLight: Bool *// Light/dark determination*

var contrastingColor: Color *// Black or white for text*

var accessibleTextColor: Color *// WCAG compliant text color*

**Color Manipulation:**

swift

func lightened(by percentage: Double) -> Color *// Lighten color*

func darkened(by percentage: Double) -> Color *// Darken color*

func adjustingSaturation(by percentage: Double) -> Color

func blended(with other: Color, ratio: Double) -> Color

var complementaryColor: Color *// Color wheel opposite*

**Advanced Gradient Creation:**

swift

func asLinearGradient(to endColor: Color?, startPoint: UnitPoint, endPoint: UnitPoint) -> LinearGradient

func asRadialGradient(to endColor: Color?, center: UnitPoint, startRadius: CGFloat, endRadius: CGFloat) -> RadialGradient

func asAngularGradient(to endColor: Color?, center: UnitPoint, startAngle: Angle, endAngle: Angle) -> AngularGradient

**Accessibility and Contrast:**

swift

func getContrastRatio(with other: Color) -> Double

func meetsContrastRequirements(with other: Color, level: ContrastLevel = .aa) -> Bool

public enum ContrastLevel {

case aa *// 4.5:1 contrast ratio*

case aaa *// 7:1 contrast ratio*

case aaLarge *// 3:1 contrast ratio for large text*

}

**Color Harmony and Theory:**

swift

func generateHarmoniousScheme(type: ColorSchemeType = .monochromatic) -> [Color]

public enum ColorSchemeType {

case monochromatic, analogous, complementary, triadic

}

var analogousColors: [Color] *// Adjacent on color wheel*

var triadicColors: [Color] *// Evenly spaced on color wheel*

var monochromaticColors: [Color] *// Same hue, different saturation/brightness*

var splitComplementaryColors: [Color] *// Split-complementary harmony*

**Material Design Integration:**

swift

struct MaterialColors {

static let red50, red100, red500, red900: Color *// Material color scales*

static let blue50, blue100, blue500, blue900: Color

static let green50, green100, green500, green900: Color

static let gray50, gray100, gray200, ..., gray900: Color

}

func materialVariants() -> [Color] *// Generate Material Design variants*

**Advanced Color Operations:**

swift

func mixed(with other: Color, mode: BlendMode, intensity: Double = 1.0) -> Color

func filtered(with filter: ColorFilter, intensity: Double = 1.0) -> Color

public enum BlendMode {

case normal, multiply, screen, overlay

}

public enum ColorFilter {

case sepia, grayscale, invert, brighten, darken

}

**Testing and Validation:**

swift

#if DEBUG

func validateAccessibility() -> AccessibilityReport

static func testColor(brightness: Double, saturation: Double = 1.0) -> Color

var debugDescription: String *// Comprehensive color information*

#endif

public struct AccessibilityReport {

public let contrastWithBlack: Double, contrastWithWhite: Double

public let meetsAA: Bool, meetsAAA: Bool

public let issues: [String], recommendations: [String]

public var bestTextColor: Color

}

**Integration Points:**

* Foundation for ThemeManager color system
* Used by ColorConstants for app-wide color definitions
* Provides accessibility validation for all color combinations
* Supports advanced color theory for harmonious palettes

## Section 52: Utils/Extensions/ViewExtensions.swift

**File Type:** SwiftUI View Enhancement System  
**Purpose:** Comprehensive view modifiers for error handling, styling, performance, and accessibility

**Error Handling Extensions:**

swift

extension View {

func errorAlert(onRetry: (() -> Void)? = nil) -> some View

func errorHandling(context: String?, showInline: Bool = false, onRetry: (() -> Void)?, onDismiss: (() -> Void)?) -> some View

func handleErrors(context: String? = nil) -> some View

func errorToast() -> some View

}

**Loading and State Extensions:**

swift

func loadingOverlay(\_ isLoading: Bool, message: String = "Loading...", style: LoadingStyle = .spinner) -> some View

func refreshable(action: @escaping () async -> Void) -> some View

func emptyState(isEmpty: Bool, title: String, message: String, systemImage: String = "tray", action: (() -> Void)?, actionTitle: String = "Retry") -> some View

func skeletonLoading(\_ isLoading: Bool, lines: Int = 3, animated: Bool = true) -> some View

**Keyboard and Input Extensions:**

swift

func dismissKeyboardOnTap() -> some View

func keyboardToolbar(onDone: @escaping () -> Void = {}) -> some View

func onKeyboardChange(perform action: @escaping (Bool, CGFloat) -> Void) -> some View

**Styling and Appearance Extensions:**

swift

func cardStyle(backgroundColor: Color, cornerRadius: CGFloat, shadowRadius: CGFloat, shadowOpacity: Float, padding: CGFloat) -> some View

func glassMorphism(blur: CGFloat = 20, opacity: Double = 0.3, cornerRadius: CGFloat) -> some View

func neumorphism(cornerRadius: CGFloat, distance: CGFloat = 6, intensity: CGFloat = 0.15) -> some View

func gradientBorder(colors: [Color], width: CGFloat = 2, cornerRadius: CGFloat) -> some View

func shimmer(active: Bool = true) -> some View

func bounceAnimation(trigger: Binding<Bool>, scale: CGFloat = 1.2, duration: Double = 0.6) -> some View

func pulseAnimation(active: Bool = true, scale: CGFloat = 1.05, duration: Double = 1.0) -> some View

**Conditional Modifiers:**

swift

@ViewBuilder

func `if`<Content: View>(\_ condition: Bool, transform: (Self) -> Content) -> some View

@ViewBuilder

func `if`<TrueContent: View, FalseContent: View>(\_ condition: Bool, ifTrue: (Self) -> TrueContent, ifFalse: (Self) -> FalseContent) -> some View

@ViewBuilder

func ifLet<Value, Content: View>(\_ optionalValue: Value?, transform: (Self, Value) -> Content) -> some View

func iPhone(\_ modifier: (Self) -> some View) -> some View

func iPad(\_ modifier: (Self) -> some View) -> some View

**Accessibility Extensions:**

swift

func accessibilityConfiguration(label: String?, hint: String?, value: String?, traits: AccessibilityTraits = [], identifier: String?) -> some View

func semanticAccessibility(role: AccessibilityRole, label: String, hint: String?, isEnabled: Bool = true) -> some View

func voiceOverAccessible(label: String, hint: String?, sortPriority: Double = 0) -> some View

func accessibilityActions(\_ actions: [AccessibilityAction]) -> some View

**Performance and Optimization Extensions:**

swift

func performanceMonitored(identifier: String, threshold: TimeInterval = 0.1) -> some View

func listOptimized() -> some View

func cached<Key: Hashable>(key: Key, computation: @escaping () -> some View) -> some View

func lazyLoaded(threshold: CGFloat = 100, placeholder: AnyView) -> some View

**Animation Extensions:**

swift

func springAnimation(preset: SpringPreset = .default, value: some Equatable) -> some View

func staggeredAnimation(delay: Double, duration: Double = 0.5) -> some View

func parallaxScroll(offsetMultiplier: CGFloat = 0.5) -> some View

func customTransition(\_ transition: AnyTransition, isVisible: Bool) -> some View

public enum SpringPreset {

case `default`, bouncy, smooth, snappy

var animation: Animation

}

**Data and State Extensions:**

swift

func asyncData<T>(\_ data: AsyncData<T>, onRetry: @escaping () -> Void, @ViewBuilder content: @escaping (T) -> some View) -> some View

func bindingWithValidation<T>(\_ binding: Binding<T>, validation: @escaping (T) -> ValidationResult) -> some View

func autoSave<T: Codable>(\_ value: T, key: String, debounceTime: TimeInterval = 1.0) -> some View

**Gesture Extensions:**

swift

func tapWithFeedback(style: UIImpactFeedbackGenerator.FeedbackStyle = .light, action: @escaping () -> Void) -> some View

func longPressWithFeedback(minimumDuration: Double = 0.5, maximumDistance: CGFloat = 10, action: @escaping () -> Void) -> some View

func swipeGestures(onLeft: (() -> Void)?, onRight: (() -> Void)?, onUp: (() -> Void)?, onDown: (() -> Void)?) -> some View

func pullToRefresh(coordinateSpace: CoordinateSpace, onRefresh: @escaping () async -> Void) -> some View

**Integration Points:**

* Used throughout the app for consistent UI behavior
* Provides standardized error handling across all features
* Integrates with performance monitoring system
* Supports accessibility requirements automatically

**Constants (Utils/Constants/)**

**53. Utils/Constants/AppConstants.swift**

**File Type:** Global Configuration System  
**Purpose:** Centralized configuration with feature flags, validation rules, and runtime constants

**Architecture:**

swift

public enum AppConstants {

*// Feature flags for functionality control*

public enum Features {

public static let enableWidgets = true

public static let enableCloudSync = false

public static let enableBiometrics = true

public static let enableDataExport = true

public static let enablePushNotifications = true

#if DEBUG

public static let enableDebugLogging = true

#else

public static let enableDebugLogging = false

#endif

}

}

**UI Configuration:**

swift

public enum UI {

public static let cornerRadius: CGFloat = 12

public static let defaultPadding: CGFloat = 16

public static let defaultAnimationDuration: Double = 0.3

public static let minimumTouchHeight: CGFloat = 44

public static let maximumButtonWidth: CGFloat = 280

public static let standardSpacing: CGFloat = 8

public static let minimumContentWidth: CGFloat = 320

public static let maximumContentWidth: CGFloat = 414

public static let defaultShadowRadius: CGFloat = 4

public static let defaultShadowOpacity: Float = 0.1

}

**Data and Validation Constants:**

swift

public enum Data {

public static let maxBudgetCategories = 20

public static let maxTransactionNoteLength = 500

public static let csvExportDateFormat = "yyyy-MM-dd"

public static let defaultCurrency = "USD"

public static let maxImportFileSize: Int64 = 10\_485\_760 *// 10MB*

public static let supportedImportFormats = ["csv"]

public static let maxRecentTransactions = 5

public static let monthsToKeepInHistory = 24

}

public enum Validation {

public static let minimumTransactionAmount: Double = 0.01

public static let maximumTransactionAmount: Double = 999999.99

public static let maxCategoryNameLength = 30

public static let minimumPasswordLength = 8

public static let maxLoginAttempts = 3

}

**Time and Storage Configuration:**

swift

public enum Time {

public static let defaultReminderHour = 20 *// 8 PM*

public static let defaultBudgetUpdateDay = 1

public static let budgetWidgetRefreshInterval: TimeInterval = 3600

public static let minimumNotificationInterval: TimeInterval = 300

public static let cacheExpirationInterval: TimeInterval = 86400

}

public enum Storage {

public static let appGroupIdentifier = "group.com.brandontitensor.BrandonsBudget"

public static let userDefaultsKeyPrefix = "com.brandontitensor.BrandonsBudget."

public static let keychainKeyPrefix = "com.brandontitensor.BrandonsBudget."

public static let databaseFilename = "BudgetModel.sqlite"

}

**URLs and External Integration:**

swift

public enum URLs {

public static let apiBaseURL = URL(string: "https://api.example.com")!

public static let privacyPolicy = URL(string: "https://www.example.com/privacy")!

public static let termsOfService = URL(string: "https://www.example.com/terms")!

public static let support = URL(string: "https://www.example.com/support")!

public static let appStore = URL(string: "https://apps.apple.com/app/id123456789")!

}

**Error Messages and Localization:**

swift

public enum ErrorMessages {

public static let genericError = NSLocalizedString("An unexpected error occurred. Please try again.", comment: "Generic error message")

public static let networkError = NSLocalizedString("Unable to connect. Please check your internet connection.", comment: "Network error message")

*// ... comprehensive error message definitions*

}

**Default Categories and Analytics:**

swift

public enum DefaultCategories {

public static let all: [String] = [

"Housing", "Transportation", "Food", "Utilities", "Insurance",

"Healthcare", "Savings", "Entertainment", "Personal Care", "Education"

]

public static let required: [String] = [

"Uncategorized", "Housing", "Food", "Utilities"

]

}

public enum AnalyticsEvents {

public static let addTransaction = "add\_transaction"

public static let updateBudget = "update\_budget"

public static let exportData = "export\_data"

public static let importData = "import\_data"

public static let viewReport = "view\_report"

}

**Integration Points:**

* Referenced throughout the app for consistent behavior
* Provides feature flags for conditional functionality
* Centralizes validation rules and limits
* Supports localization and configuration management

**54. Utils/Constants/ColorConstants.swift**

**File Type:** Comprehensive Color System  
**Purpose:** Centralized color definitions with accessibility validation and dynamic theming support

**Architecture:**

swift

public enum ColorConstants {

*// Brand color hierarchy*

public enum Brand {

public static let primary = Color("BrandPrimary", bundle: .main) ?? Color.blue

public static let secondary = Color("BrandSecondary", bundle: .main) ?? Color.indigo

public static let tertiary = Color("BrandTertiary", bundle: .main) ?? Color.cyan

public enum Variants {

public static let light = Brand.primary.lightened(by: 20)

public static let dark = Brand.primary.darkened(by: 20)

public static let muted = Brand.primary.opacity(0.6)

public static let subtle = Brand.primary.opacity(0.1)

}

}

}

**Semantic Color System:**

swift

public enum Semantic {

*// Status colors with specific RGB values*

public static let success = Color(red: 34/255, green: 197/255, blue: 94/255)

public static let warning = Color(red: 251/255, green: 191/255, blue: 36/255)

public static let error = Color(red: 239/255, green: 68/255, blue: 68/255)

public static let info = Color(red: 59/255, green: 130/255, blue: 246/255)

*// Budget-specific semantic colors*

public static let income = Color(red: 16/255, green: 185/255, blue: 129/255)

public static let expense = Color(red: 220/255, green: 38/255, blue: 127/255)

public static let savings = Color(red: 99/255, green: 102/255, blue: 241/255)

public static let investment = Color(red: 168/255, green: 85/255, blue: 247/255)

public enum Variants {

*// Semantic color variants for different contexts*

public static let successLight = Semantic.success.lightened(by: 30)

public static let successDark = Semantic.success.darkened(by: 20)

public static let successSubtle = Semantic.success.opacity(0.1)

*// ... complete variant system*

}

}

**Neutral Color System:**

swift

public enum Neutral {

*// Grayscale palette with precise RGB values*

public static let white = Color.white

public static let gray50 = Color(red: 249/255, green: 250/255, blue: 251/255)

public static let gray100 = Color(red: 243/255, green: 244/255, blue: 246/255)

*// ... complete grayscale system through gray900*

public static let black = Color.black

*// System-aware colors*

public static let background = Color(.systemBackground)

public static let secondaryBackground = Color(.secondarySystemBackground)

public static let text = Color(.label)

public static let secondaryText = Color(.secondaryLabel)

*// Functional colors*

public static let border = gray300

public static let focusedBorder = Brand.primary

public static let errorBorder = Semantic.error

}

**Category Color System:**

swift

public enum Category {

private static let categoryPalette: [Color] = [

Color(red: 239/255, green: 68/255, blue: 68/255), *// Red*

Color(red: 34/255, green: 197/255, blue: 94/255), *// Green*

Color(red: 59/255, green: 130/255, blue: 246/255), *// Blue*

*// ... 16 total harmonious colors*

]

*/// Get consistent color for category name using stable hash*

public static func color(for categoryName: String) -> Color

public static func lightColor(for categoryName: String) -> Color

public static func subtleColor(for categoryName: String) -> Color

public static var allColors: [Color]

}

**Chart and Visualization Colors:**

swift

public enum Chart {

public static let palette: [Color] = [

primary, secondary, tertiary, quaternary,

Color(red: 239/255, green: 68/255, blue: 68/255), *// Red*

Color(red: 6/255, green: 182/255, blue: 212/255), *// Cyan*

*// ... 8 total distinct colors*

]

public enum PieChart {

public static let colors = Chart.palette

public static let strokeColor = Neutral.white

public static let strokeWidth: CGFloat = 2

}

public static func color(at index: Int) -> Color *// Cycling access*

}

**Gradient Definitions:**

swift

public enum Gradients {

public static let primary = LinearGradient(

colors: [Brand.primary, Brand.secondary],

startPoint: .topLeading, endPoint: .bottomTrailing

)

public static let success = LinearGradient(

colors: [Semantic.success, Semantic.success.darkened(by: 20)],

startPoint: .topLeading, endPoint: .bottomTrailing

)

*// Custom gradient generators*

public static func radial(for color: Color) -> RadialGradient

public static func angular(colors: [Color]) -> AngularGradient

}

**Accessibility Color System:**

swift

public enum Accessibility {

public static let highContrastText = Neutral.black

public static let highContrastBackground = Neutral.white

public static let focusRing = Brand.primary

public static let focusRingWidth: CGFloat = 3

*// Error states with WCAG compliance*

public static let errorText = Color(red: 153/255, green: 27/255, blue: 27/255)

public static let errorBackground = Color(red: 254/255, green: 242/255, blue: 242/255)

*// Validation methods*

public static func meetsAAStandards(foreground: Color, background: Color) -> Bool

public static func meetsAAAStandards(foreground: Color, background: Color) -> Bool

public static func bestTextColor(for backgroundColor: Color) -> Color

}

**Dynamic Color Support:**

swift

public enum Dynamic {

public static func color(light: Color, dark: Color) -> Color

*// Pre-defined dynamic colors*

public static let primaryText = color(light: Neutral.text, dark: DarkMode.text)

public static let background = color(light: Neutral.background, dark: DarkMode.background)

}

public enum DarkMode {

public static let primaryLight = Brand.primary.lightened(by: 20)

public static let background = Color(red: 17/255, green: 24/255, blue: 39/255)

public static let text = Color(red: 243/255, green: 244/255, blue: 246/255)

*// ... complete dark mode color system*

}

**Utility Functions:**

swift

public enum Utils {

public static func randomCategoryColor() -> Color

public static func accessibleColor(\_ color: Color, on background: Color) -> Color

public static func validatePalette(\_ colors: [Color]) -> [PaletteValidationResult]

public static func colorFromHex(\_ hex: String) -> Color?

public static func tint(\_ color: Color, by percentage: Double) -> Color

public static func shade(\_ color: Color, by percentage: Double) -> Color

}

**Testing Support:**

swift

#if DEBUG

public extension ColorConstants {

enum Testing {

static func allTestColors() -> [String: Color]

static func testAccessibility() -> [String: Bool]

static func colorPalettePreview() -> some View

static func testCategoryColorConsistency() -> Bool

}

}

#endif

**Integration Points:**

* Foundation for all app theming and color decisions
* Provides accessibility-validated color combinations
* Supports dynamic theming and dark mode
* Used by ThemeManager for color coordination

**Helpers (Utils/Helpers/)**

# Section 55: Utils/Helpers/FormatHelpers.swift

**File Type:** Comprehensive Formatting System **Purpose:** Centralized formatting utilities with localization, validation, and performance optimization

## Architecture Overview:

swift

*/// Collection of formatting utilities for the app with proper localization and validation*

public enum FormatHelpers {

*// MARK: - Currency Formatting*

public struct CurrencyFormatter {

private let formatter: NumberFormatter

private let locale: Locale

private let currencyCode: String

public init(currencyCode: String = "USD", locale: Locale = .current)

private mutating func setupFormatter()

}

*// MARK: - Date Formatting*

public struct DateFormatter {

private static let cache = NSCache<NSString, Foundation.DateFormatter>()

private static func cachedFormatter(format: String, locale: Locale = .current) -> Foundation.DateFormatter

}

*// MARK: - Percentage Formatting*

public struct PercentageFormatter {

private let formatter: NumberFormatter

public init(decimalPlaces: Int = 1)

}

*// MARK: - Number Formatting*

public struct NumberFormatter { */\* Static utility methods \*/* }

*// MARK: - Budget-Specific Formatting*

public struct BudgetFormatter { */\* Budget analysis and status formatting \*/* }

*// MARK: - Text Formatting*

public struct TextFormatter { */\* Text manipulation utilities \*/* }

*// MARK: - Validation Helpers*

public struct ValidationFormatter { */\* Input cleaning and validation \*/* }

}

## Core Currency Formatting System:

swift

public struct CurrencyFormatter {

*/// Format amount as currency string*

public func format(\_ amount: Double) -> String {

return formatter.string(from: NSNumber(value: amount)) ?? "$0.00"

}

*/// Format amount with custom decimal places*

public func format(\_ amount: Double, decimalPlaces: Int) -> String

*/// Format amount without currency symbol*

public func formatAmount(\_ amount: Double) -> String

*/// Format amount with compact notation (K, M, B)*

public func formatCompact(\_ amount: Double) -> String {

let absAmount = abs(amount)

let sign = amount < 0 ? "-" : ""

if absAmount >= 1\_000\_000\_000 {

return "\(sign)\(format(absAmount / 1\_000\_000\_000, decimalPlaces: 1))B"

} else if absAmount >= 1\_000\_000 {

return "\(sign)\(format(absAmount / 1\_000\_000, decimalPlaces: 1))M"

} else if absAmount >= 1\_000 {

return "\(sign)\(format(absAmount / 1\_000, decimalPlaces: 1))K"

} else {

return format(amount)

}

}

}

## Advanced Date Formatting with Caching:

swift

public struct DateFormatter {

*/// Get cached formatter for performance*

private static func cachedFormatter(format: String, locale: Locale = .current) -> Foundation.DateFormatter {

let key = "\(format)\_\(locale.identifier)" as NSString

if let cached = cache.object(forKey: key) {

return cached

}

let formatter = Foundation.DateFormatter()

formatter.dateFormat = format

formatter.locale = locale

formatter.timeZone = TimeZone.current

cache.setObject(formatter, forKey: key)

return formatter

}

*/// Format date for transactions (e.g., "Dec 15")*

public static func formatTransaction(\_ date: Date) -> String {

return cachedFormatter(format: "MMM d").string(from: date)

}

*/// Format date for budget periods (e.g., "December 2024")*

public static func formatBudgetPeriod(\_ date: Date) -> String {

return cachedFormatter(format: "MMMM yyyy").string(from: date)

}

*/// Format relative date (e.g., "2 days ago", "in 3 hours")*

public static func formatRelative(\_ date: Date, relativeTo referenceDate: Date = Date()) -> String {

let formatter = RelativeDateTimeFormatter()

formatter.dateTimeStyle = .named

formatter.unitsStyle = .full

return formatter.localizedString(for: date, relativeTo: referenceDate)

}

*/// Format timestamp for files (e.g., "20241215\_143022")*

public static func formatTimestamp(\_ date: Date) -> String {

return cachedFormatter(format: "yyyyMMdd\_HHmmss").string(from: date)

}

}

## Budget-Specific Formatting:

swift

public struct BudgetFormatter {

*/// Format budget status with color coding*

public static func formatBudgetStatus(

spent: Double,

budget: Double,

currency: String = "USD"

) -> (text: String, color: String) {

let currencyFormatter = CurrencyFormatter(currencyCode: currency)

let percentage = budget > 0 ? (spent / budget) \* 100 : 0

let text: String

let color: String

if spent > budget {

let over = spent - budget

text = "Over by \(currencyFormatter.format(over))"

color = "red"

} else if percentage > 90 {

let remaining = budget - spent

text = "\(currencyFormatter.format(remaining)) left"

color = "orange"

} else if percentage > 75 {

let remaining = budget - spent

text = "\(currencyFormatter.format(remaining)) remaining"

color = "yellow"

} else {

let remaining = budget - spent

text = "\(currencyFormatter.format(remaining)) available"

color = "green"

}

return (text, color)

}

*/// Format spending trend*

public static func formatSpendingTrend(

current: Double,

previous: Double,

currency: String = "USD"

) -> (text: String, isIncrease: Bool) {

let currencyFormatter = CurrencyFormatter(currencyCode: currency)

let difference = current - previous

let isIncrease = difference > 0

if abs(difference) < 0.01 {

return ("No change", false)

}

let percentageChange = previous > 0 ? abs(difference / previous) \* 100 : 0

let arrow = isIncrease ? "↑" : "↓"

let verb = isIncrease ? "increase" : "decrease"

let text = "\(arrow) \(currencyFormatter.format(abs(difference))) (\(String(format: "%.1f", percentageChange))% \(verb))"

return (text, isIncrease)

}

}

## Input Validation and Cleaning:

swift

public struct ValidationFormatter {

*/// Clean and validate currency input*

public static func cleanCurrencyInput(\_ input: String) -> String {

let cleaned = input

.replacingOccurrences(of: "[^0-9.,]", with: "", options: .regularExpression)

.replacingOccurrences(of: ",", with: "")

*// Handle multiple decimal points*

let components = cleaned.components(separatedBy: ".")

if components.count > 2 {

return components[0] + "." + components[1]

}

return cleaned

}

*/// Validate and format percentage input*

public static func cleanPercentageInput(\_ input: String) -> String {

let cleaned = input

.replacingOccurrences(of: "[^0-9.]", with: "", options: .regularExpression)

if let value = Double(cleaned), value > 100 {

return "100"

}

return cleaned

}

}

## Extensions for Convenience:

swift

extension Double {

*/// Format as currency using default formatter*

public var formattedAsCurrency: String {

return FormatHelpers.CurrencyFormatter().format(self)

}

*/// Format as percentage*

public var formattedAsPercentage: String {

return FormatHelpers.PercentageFormatter().format(self)

}

}

extension Date {

*/// Format for transaction display*

public var formattedForTransaction: String {

return FormatHelpers.DateFormatter.formatTransaction(self)

}

*/// Format relative to now*

public var formattedRelative: String {

return FormatHelpers.DateFormatter.formatRelative(self)

}

}

extension String {

*/// Clean for currency input*

public var cleanedForCurrency: String {

return FormatHelpers.ValidationFormatter.cleanCurrencyInput(self)

}

*/// Format as title case*

public var titleCased: String {

return FormatHelpers.TextFormatter.titleCase(self)

}

}

**Integration Points:**

* Used throughout app for consistent data formatting
* Provides cached formatters for performance optimization
* Supports localization and multiple currency codes
* Integrates with validation systems for input cleaning

# Section 56: Utils/Helpers/PerformanceMonitor.swift

**File Type:** Advanced Performance Monitoring System **Purpose:** Comprehensive performance tracking with metrics, analytics, and system health monitoring

## Architecture Overview:

swift

@MainActor

public final class PerformanceMonitor: ObservableObject {

public static let shared = PerformanceMonitor()

*// MARK: - Published Properties*

@Published public private(set) var isMonitoring = false

@Published public private(set) var currentMetrics: [PerformanceMetric] = []

@Published public private(set) var systemMetrics: SystemMetrics?

@Published public private(set) var slowOperations: [PerformanceMetric] = []

@Published public private(set) var memoryWarnings: [Date] = []

@Published public private(set) var overallPerformanceScore: Double = 1.0

*// MARK: - Private Properties*

private var activeOperations: [String: Date] = [:]

private var metricsHistory: [PerformanceMetric] = []

private var systemMetricsHistory: [SystemMetrics] = []

private let metricsQueue = DispatchQueue(label: "com.brandonsbudget.performance", qos: .utility)

private var cancellables = Set<AnyCancellable>()

private var systemTimer: Timer?

private var cleanupTimer: Timer?

}

## Core Performance Metric System:

swift

public struct PerformanceMetric {

public let operation: String

public let duration: TimeInterval

public let timestamp: Date

public let context: String?

public let memoryUsage: UInt64?

public let thread: String

*/// Human readable duration*

public var formattedDuration: String {

if duration >= 1.0 {

return String(format: "%.2fs", duration)

} else {

return String(format: "%.0fms", duration \* 1000)

}

}

*/// Performance rating based on duration*

public var performanceRating: PerformanceRating {

if duration < 0.1 {

return .excellent

} else if duration < 0.5 {

return .good

} else if duration < 1.0 {

return .fair

} else {

return .poor

}

}

}

public enum PerformanceRating: String, CaseIterable {

case excellent = "Excellent"

case good = "Good"

case fair = "Fair"

case poor = "Poor"

public var color: Color {

switch self {

case .excellent: return .green

case .good: return .blue

case .fair: return .orange

case .poor: return .red

}

}

}

## System Metrics Monitoring:

swift

public struct SystemMetrics {

public let memoryUsage: UInt64

public let memoryPressure: MemoryPressure

public let cpuUsage: Double

public let diskSpace: DiskSpace

public let batteryLevel: Float

public let thermalState: ProcessInfo.ThermalState

public let timestamp: Date

public enum MemoryPressure: String, CaseIterable {

case normal = "Normal"

case warning = "Warning"

case critical = "Critical"

public var color: Color {

switch self {

case .normal: return .green

case .warning: return .orange

case .critical: return .red

}

}

}

public struct DiskSpace {

public let available: UInt64

public let total: UInt64

public var usedPercentage: Double {

let used = total - available

return Double(used) / Double(total) \* 100

}

public var availableFormatted: String {

return ByteCountFormatter.string(fromByteCount: Int64(available), countStyle: .file)

}

}

}

## Core Monitoring Operations:

swift

*/// Start timing an operation*

public func startTiming(\_ operation: String, context: String? = nil) {

guard isMonitoring else { return }

metricsQueue.async {

let key = context != nil ? "\(operation)\_\(context!)" : operation

self.activeOperations[key] = Date()

}

}

*/// End timing an operation and return duration*

@discardableResult

public func endTiming(\_ operation: String, context: String? = nil) -> TimeInterval? {

guard isMonitoring else { return nil }

let endTime = Date()

var duration: TimeInterval?

metricsQueue.sync {

let key = context != nil ? "\(operation)\_\(context!)" : operation

guard let startTime = activeOperations.removeValue(forKey: key) else {

return

}

duration = endTime.timeIntervalSince(startTime)

let metric = PerformanceMetric(

operation: operation,

duration: duration!,

timestamp: endTime,

context: context,

memoryUsage: getCurrentMemoryUsage()

)

metricsHistory.append(metric)

*// Track slow operations*

if duration! >= slowOperationThreshold {

Task { @MainActor in

slowOperations.append(metric)

if slowOperations.count > 10 {

slowOperations.removeFirst()

}

}

}

}

return duration

}

*/// Measure execution time of a closure*

@discardableResult

public func measure<T>(\_ operation: String, context: String? = nil, \_ closure: () throws -> T) rethrows -> T {

startTiming(operation, context: context)

defer { endTiming(operation, context: context) }

return try closure()

}

## Performance Reporting System:

swift

public struct PerformanceReport {

public let timeRange: DateInterval

public let metrics: [PerformanceMetric]

public let systemMetrics: [SystemMetrics]

public let slowestOperations: [PerformanceMetric]

public let averageDurations: [String: TimeInterval]

public let performanceScore: Double

public let recommendations: [String]

public let generatedAt: Date

public var summary: String {

let score = Int(performanceScore \* 100)

return "Performance Score: \(score)% (\(metrics.count) operations tracked)"

}

public var overallRating: PerformanceRating {

if performanceScore >= 0.9 { return .excellent }

if performanceScore >= 0.7 { return .good }

if performanceScore >= 0.5 { return .fair }

return .poor

}

}

*/// Get performance report for a time period*

public func getPerformanceReport(for period: TimePeriod = .last7Days) -> PerformanceReport {

let dateInterval = period.dateInterval()

return metricsQueue.sync {

let filteredMetrics = metricsHistory.filter { metric in

dateInterval.contains(metric.timestamp)

}

let slowest = filteredMetrics

.sorted { $0.duration > $1.duration }

.prefix(10)

.map { $0 }

let averages = Dictionary(grouping: filteredMetrics, by: \.operation)

.mapValues { metrics in

metrics.reduce(0) { $0 + $1.duration } / Double(metrics.count)

}

let score = calculatePerformanceScore(for: filteredMetrics)

let recommendations = generateRecommendations(for: filteredMetrics, systemMetrics: systemMetricsHistory)

return PerformanceReport(

timeRange: dateInterval,

metrics: filteredMetrics,

systemMetrics: systemMetricsHistory,

slowestOperations: slowest,

averageDurations: averages,

performanceScore: score,

recommendations: recommendations,

generatedAt: Date()

)

}

}

## System Health Monitoring:

swift

private func getCurrentMemoryUsage() -> UInt64 {

var info = mach\_task\_basic\_info()

var count = mach\_msg\_type\_number\_t(MemoryLayout<mach\_task\_basic\_info>.size) / 4

let result = withUnsafeMutablePointer(to: &info) {

$0.withMemoryRebound(to: integer\_t.self, capacity: 1) {

task\_info(mach\_task\_self\_, task\_flavor\_t(MACH\_TASK\_BASIC\_INFO), $0, &count)

}

}

return result == KERN\_SUCCESS ? info.resident\_size : 0

}

private func setupMemoryWarningObserver() {

NotificationCenter.default.publisher(for: UIApplication.didReceiveMemoryWarningNotification)

.sink { [weak self] \_ in

Task { @MainActor [weak self] in

self?.memoryWarnings.append(Date())

if let self = self, self.memoryWarnings.count > 10 {

self.memoryWarnings.removeFirst()

}

}

print("⚠️ PerformanceMonitor: Memory warning received")

}

.store(in: &cancellables)

}

## Debug and Testing Support:

swift

#if DEBUG

extension PerformanceMonitor {

*/// Generate test performance data*

public func generateTestData() {

let operations = ["DataLoad", "UIUpdate", "NetworkRequest", "DatabaseQuery", "ImageProcessing"]

let contexts = ["UserAction", "BackgroundSync", "AppLaunch", "SettingsChange"]

for \_ in 0..<50 {

let operation = operations.randomElement()!

let context = contexts.randomElement()!

let duration = TimeInterval.random(in: 0.01...2.0)

let metric = PerformanceMetric(

operation: operation,

duration: duration,

context: context,

memoryUsage: UInt64.random(in: 50\_000\_000...200\_000\_000)

)

metricsHistory.append(metric)

}

}

*/// Get performance insights for debugging*

public func getDebugInsights() -> [String: Any] {

return metricsQueue.sync {

let totalMetrics = metricsHistory.count

let slowMetrics = metricsHistory.filter { $0.duration > slowOperationThreshold }.count

let averageDuration = metricsHistory.isEmpty ? 0 : metricsHistory.reduce(0) { $0 + $1.duration } / Double(metricsHistory.count)

return [

"totalMetrics": totalMetrics,

"slowMetrics": slowMetrics,

"averageDuration": averageDuration,

"memoryWarnings": memoryWarnings.count,

"overallScore": overallPerformanceScore,

"isMonitoring": isMonitoring

]

}

}

}

#endif

**Integration Points:**

* Automatic monitoring of all major app operations
* Integration with error handling for performance-related issues
* System health monitoring with proactive warnings
* Analytics and reporting for performance optimization

# Section 57: Utils/Helpers/ValidationHelpers.swift

**File Type:** Comprehensive Validation System **Purpose:** Type-safe validation with detailed error handling and recovery suggestions

## Validation Architecture:

swift

*/// Represents the result of a validation operation*

public enum ValidationResult: Equatable {

case valid

case invalid(ValidationError)

*/// Whether the validation passed*

public var isValid: Bool {

if case .valid = self { return true }

return false

}

*/// The error if validation failed*

public var error: ValidationError? {

if case .invalid(let error) = self { return error }

return nil

}

*/// User-friendly error message*

public var errorMessage: String? {

return error?.localizedDescription

}

}

## Comprehensive Error Types:

swift

*/// Comprehensive validation error types*

public enum ValidationError: LocalizedError, Equatable {

*// String validation errors*

case empty(field: String)

case tooShort(field: String, minimum: Int, actual: Int)

case tooLong(field: String, maximum: Int, actual: Int)

case invalidFormat(field: String, format: String)

case containsInvalidCharacters(field: String, characters: String)

case notAllowed(field: String, value: String, reason: String)

*// Numeric validation errors*

case notANumber(field: String, value: String)

case tooSmall(field: String, minimum: Double, actual: Double)

case tooLarge(field: String, maximum: Double, actual: Double)

case notInteger(field: String, value: Double)

case negativeNotAllowed(field: String, value: Double)

case zeroNotAllowed(field: String)

*// Date validation errors*

case invalidDate(field: String, value: String)

case dateInPast(field: String, date: Date)

case dateInFuture(field: String, date: Date)

case dateOutOfRange(field: String, date: Date, min: Date?, max: Date?)

*// Email validation errors*

case invalidEmail(email: String)

case emailTooLong(email: String, maxLength: Int)

case emailDomainInvalid(email: String, domain: String)

*// Budget-specific validation errors*

case invalidCurrency(amount: String)

case budgetExceeded(category: String, amount: Double, limit: Double)

case categoryNotFound(category: String)

case duplicateCategory(category: String)

case invalidTransactionDate(date: Date, reason: String)

case invalidBudgetPeriod(period: String)

*// Custom validation errors*

case custom(message: String)

case multipleErrors([ValidationError])

public var errorDescription: String? {

switch self {

case .empty(let field):

return "\(field) cannot be empty"

case .tooShort(let field, let minimum, let actual):

return "\(field) is too short (minimum \(minimum) characters, got \(actual))"

case .invalidCurrency(let amount):

return "Invalid currency amount: \(amount)"

case .budgetExceeded(let category, let amount, let limit):

return "\(category) budget exceeded: \(amount.asCurrency) over limit of \(limit.asCurrency)"

*// ... complete error descriptions for all cases*

}

}

public var recoverySuggestion: String? {

switch self {

case .empty:

return "Please enter a value for this field"

case .tooShort(\_, let minimum, \_):

return "Please enter at least \(minimum) characters"

case .invalidEmail:

return "Please enter a valid email address (e.g., user@example.com)"

case .budgetExceeded:

return "Consider adjusting your budget or reducing the amount"

*// ... complete recovery suggestions for all cases*

}

}

}

## Core Validation Methods:

swift

public enum ValidationHelpers {

*/// Validate that a string is not empty*

public static func validateNotEmpty(\_ value: String, fieldName: String = "Field") -> ValidationResult {

let trimmed = value.trimmingCharacters(in: .whitespacesAndNewlines)

return trimmed.isEmpty ? .invalid(.empty(field: fieldName)) : .valid

}

*/// Validate string length*

public static func validateStringLength(

\_ value: String,

fieldName: String = "Field",

minLength: Int? = nil,

maxLength: Int? = nil

) -> ValidationResult {

let trimmed = value.trimmingCharacters(in: .whitespacesAndNewlines)

if let min = minLength, trimmed.count < min {

return .invalid(.tooShort(field: fieldName, minimum: min, actual: trimmed.count))

}

if let max = maxLength, trimmed.count > max {

return .invalid(.tooLong(field: fieldName, maximum: max, actual: trimmed.count))

}

return .valid

}

*/// Validate that a string represents a valid number*

public static func validateNumber(\_ value: String, fieldName: String = "Field") -> (ValidationResult, Double?) {

let trimmed = value.trimmingCharacters(in: .whitespacesAndNewlines)

*// Remove currency symbols and formatting*

let cleanedValue = trimmed

.replacingOccurrences(of: "$", with: "")

.replacingOccurrences(of: ",", with: "")

.replacingOccurrences(of: " ", with: "")

guard let number = Double(cleanedValue) else {

return (.invalid(.notANumber(field: fieldName, value: trimmed)), nil)

}

return (.valid, number)

}

*/// Validate currency amount*

public static func validateCurrency(

\_ value: String,

fieldName: String = "Amount",

maxAmount: Double? = nil

) -> (ValidationResult, Double?) {

let (numberResult, amount) = validateNumber(value, fieldName: fieldName)

guard case .valid = numberResult, let validAmount = amount else {

return (.invalid(.invalidCurrency(amount: value)), nil)

}

let rangeResult = validateNumberRange(

validAmount,

fieldName: fieldName,

min: 0,

max: maxAmount,

allowZero: false,

allowNegative: false

)

return (rangeResult, validAmount)

}

*/// Validate email address*

public static func validateEmail(\_ email: String, maxLength: Int = 254) -> ValidationResult {

let trimmed = email.trimmingCharacters(in: .whitespacesAndNewlines)

*// Check length*

if trimmed.count > maxLength {

return .invalid(.emailTooLong(email: trimmed, maxLength: maxLength))

}

*// Basic email regex pattern*

let emailPattern = "^[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$"

do {

let regex = try NSRegularExpression(pattern: emailPattern, options: [])

let range = NSRange(location: 0, length: trimmed.utf16.count)

let matches = regex.matches(in: trimmed, options: [], range: range)

if matches.isEmpty {

return .invalid(.invalidEmail(email: trimmed))

}

return .valid

} catch {

return .invalid(.custom(message: "Email validation failed"))

}

}

*/// Validate multiple conditions and return combined result*

public static func validateAll(\_ validations: [ValidationResult]) -> ValidationResult {

let errors = validations.compactMap { result in

if case .invalid(let error) = result {

return error

}

return nil

}

if errors.isEmpty {

return .valid

} else if errors.count == 1 {

return .invalid(errors[0])

} else {

return .invalid(.multipleErrors(errors))

}

}

}

## Validation Rule Builder:

swift

*/// Builder pattern for creating complex validation rules*

public struct ValidationRuleBuilder<T> {

private var rules: [(T) -> ValidationResult] = []

public init() {}

*/// Add a validation rule*

public func addRule(\_ rule: @escaping (T) -> ValidationResult) -> ValidationRuleBuilder<T> {

var builder = self

builder.rules.append(rule)

return builder

}

*/// Add a condition-based rule*

public func addCondition(\_ condition: @escaping (T) -> Bool, error: ValidationError) -> ValidationRuleBuilder<T> {

return addRule { value in

condition(value) ? .valid : .invalid(error)

}

}

*/// Build and execute all validation rules*

public func validate(\_ value: T) -> ValidationResult {

let results = rules.map { $0(value) }

return ValidationHelpers.validateAll(results)

}

}

## Pre-built Validation Rules:

swift

*/// Pre-built validation rules for common scenarios*

public enum CommonValidationRules {

*/// User name validation rules*

public static let userName = ValidationRuleBuilder<String>()

.addRule { ValidationHelpers.validateNotEmpty($0, fieldName: "Username") }

.addRule { ValidationHelpers.validateStringLength($0, fieldName: "Username", minLength: 2, maxLength: 30) }

.addRule { ValidationHelpers.validateAllowedCharacters($0, fieldName: "Username", allowedCharacters: .alphanumerics.union(CharacterSet(charactersIn: "\_-"))) }

*/// Budget amount validation rules*

public static let budgetAmount = ValidationRuleBuilder<String>()

.addRule { value in

let (result, \_) = ValidationHelpers.validateCurrency(value, maxAmount: AppConstants.Validation.maximumTransactionAmount)

return result

}

*/// Email validation rules*

public static let email = ValidationRuleBuilder<String>()

.addRule { ValidationHelpers.validateNotEmpty($0, fieldName: "Email") }

.addRule { ValidationHelpers.validateEmail($0) }

}

## SwiftUI Integration:

swift

*/// Property wrapper for validated values in SwiftUI*

@propertyWrapper

public struct Validated<T> {

private var value: T

private let validator: (T) -> ValidationResult

private var cachedResult: ValidationResult?

public var wrappedValue: T {

get { value }

set {

value = newValue

cachedResult = nil *// Clear cache when value changes*

}

}

public var projectedValue: ValidationResult {

if let cached = cachedResult {

return cached

}

let result = validator(value)

cachedResult = result

return result

}

public init(wrappedValue: T, validator: @escaping (T) -> ValidationResult) {

self.value = wrappedValue

self.validator = validator

}

}

*/// SwiftUI view modifier for validation display*

public struct ValidationModifier: ViewModifier {

let validationResult: ValidationResult

let showValidation: Bool

public func body(content: Content) -> some View {

VStack(alignment: .leading, spacing: 4) {

Content

if showValidation, case .invalid(let error) = validationResult {

Text(error.errorDescription ?? "Invalid input")

.font(.caption)

.foregroundColor(.red)

.transition(.opacity)

}

}

}

}

public extension View {

*/// Add validation display to a view*

func validation(\_ result: ValidationResult, show: Bool = true) -> some View {

modifier(ValidationModifier(validationResult: result, showValidation: show))

}

}

## Advanced Validation Features:

swift

extension ValidationHelpers {

*/// Validate category name*

public static func validateCategory(

\_ category: String,

existingCategories: [String] = [],

maxLength: Int = AppConstants.Validation.maxCategoryNameLength

) -> ValidationResult {

*// Check if empty*

let emptyResult = validateNotEmpty(category, fieldName: "Category")

if case .invalid = emptyResult { return emptyResult }

*// Check length*

let lengthResult = validateStringLength(category, fieldName: "Category", maxLength: maxLength)

if case .invalid = lengthResult { return lengthResult }

*// Check for duplicates*

let trimmed = category.trimmingCharacters(in: .whitespacesAndNewlines)

if existingCategories.contains(where: { $0.lowercased() == trimmed.lowercased() }) {

return .invalid(.duplicateCategory(category: trimmed))

}

*// Check for valid characters (letters, numbers, spaces, hyphens, underscores)*

let allowedCharacters = CharacterSet.alphanumerics.union(.whitespaces).union(CharacterSet(charactersIn: "-\_"))

return validateAllowedCharacters(trimmed, fieldName: "Category", allowedCharacters: allowedCharacters)

}

*/// Validate transaction amount against budget*

public static func validateBudgetLimit(

amount: Double,

category: String,

currentSpent: Double,

budgetLimit: Double,

allowOverBudget: Bool = true

) -> ValidationResult {

let newTotal = currentSpent + amount

if !allowOverBudget && newTotal > budgetLimit {

let overAmount = newTotal - budgetLimit

return .invalid(.budgetExceeded(category: category, amount: overAmount, limit: budgetLimit))

}

return .valid

}

*/// Validate transaction date for budget entry*

public static func validateTransactionDate(

\_ date: Date,

allowFuture: Bool = false,

maxPastDays: Int = 365

) -> ValidationResult {

let now = Date()

let maxPastDate = Calendar.current.date(byAdding: .day, value: -maxPastDays, to: now) ?? now

if !allowFuture && date > now {

return .invalid(.invalidTransactionDate(date: date, reason: "Future dates are not allowed"))

}

if date < maxPastDate {

return .invalid(.invalidTransactionDate(date: date, reason: "Date is too far in the past"))

}

return .valid

}

*/// Validate numeric range*

public static func validateNumberRange(

\_ value: Double,

fieldName: String = "Field",

min: Double? = nil,

max: Double? = nil,

allowZero: Bool = true,

allowNegative: Bool = true

) -> ValidationResult {

if !allowNegative && value < 0 {

return .invalid(.negativeNotAllowed(field: fieldName, value: value))

}

if !allowZero && value == 0 {

return .invalid(.zeroNotAllowed(field: fieldName))

}

if let minimum = min, value < minimum {

return .invalid(.tooSmall(field: fieldName, minimum: minimum, actual: value))

}

if let maximum = max, value > maximum {

return .invalid(.tooLarge(field: fieldName, maximum: maximum, actual: value))

}

return .valid

}

*/// Validate that string contains only allowed characters*

public static func validateAllowedCharacters(

\_ value: String,

fieldName: String = "Field",

allowedCharacters: CharacterSet

) -> ValidationResult {

let trimmed = value.trimmingCharacters(in: .whitespacesAndNewlines)

if let range = trimmed.rangeOfCharacter(from: allowedCharacters.inverted) {

let invalidChars = String(trimmed[range])

return .invalid(.containsInvalidCharacters(field: fieldName, characters: invalidChars))

}

return .valid

}

*/// Async validation for operations that require network calls or database queries*

public static func validateAsync<T>(

\_ value: T,

validator: @escaping (T) async throws -> ValidationResult

) async -> ValidationResult {

do {

return try await validator(value)

} catch {

return .invalid(.custom(message: "Validation failed: \(error.localizedDescription)"))

}

}

*/// Validate with custom conditions*

public static func validateCondition(\_ condition: Bool, error: ValidationError) -> ValidationResult {

return condition ? .valid : .invalid(error)

}

}

## Date and Format Validation:

swift

extension ValidationHelpers {

*/// Validate date string*

public static func validateDate(

\_ value: String,

fieldName: String = "Date",

format: String = "yyyy-MM-dd"

) -> (ValidationResult, Date?) {

let trimmed = value.trimmingCharacters(in: .whitespacesAndNewlines)

let formatter = DateFormatter()

formatter.dateFormat = format

formatter.locale = Locale(identifier: "en\_US\_POSIX")

guard let date = formatter.date(from: trimmed) else {

return (.invalid(.invalidDate(field: fieldName, value: trimmed)), nil)

}

return (.valid, date)

}

*/// Validate date range*

public static func validateDateRange(

\_ date: Date,

fieldName: String = "Date",

minDate: Date? = nil,

maxDate: Date? = nil,

allowPast: Bool = true,

allowFuture: Bool = true

) -> ValidationResult {

let now = Date()

if !allowPast && date < now {

return .invalid(.dateInPast(field: fieldName, date: date))

}

if !allowFuture && date > now {

return .invalid(.dateInFuture(field: fieldName, date: date))

}

if let min = minDate, date < min {

return .invalid(.dateOutOfRange(field: fieldName, date: date, min: min, max: maxDate))

}

if let max = maxDate, date > max {

return .invalid(.dateOutOfRange(field: fieldName, date: date, min: minDate, max: max))

}

return .valid

}

*/// Validate string format using regex*

public static func validateStringFormat(

\_ value: String,

fieldName: String = "Field",

pattern: String,

formatDescription: String

) -> ValidationResult {

let trimmed = value.trimmingCharacters(in: .whitespacesAndNewlines)

do {

let regex = try NSRegularExpression(pattern: pattern, options: [])

let range = NSRange(location: 0, length: trimmed.utf16.count)

let matches = regex.matches(in: trimmed, options: [], range: range)

return matches.isEmpty ? .invalid(.invalidFormat(field: fieldName, format: formatDescription)) : .valid

} catch {

return .invalid(.custom(message: "Invalid validation pattern"))

}

}

}

## Testing Support:

swift

#if DEBUG

public extension ValidationHelpers {

*/// Testing utilities for validation*

enum Testing {

*/// Test all validation functions with sample data*

public static func runValidationTests() -> [String: Bool] {

var results: [String: Bool] = [:]

*// String validation tests*

results["empty\_string"] = (validateNotEmpty("", fieldName: "Test").isValid == false)

results["valid\_string"] = validateNotEmpty("Valid", fieldName: "Test").isValid

results["long\_string"] = (validateStringLength("Very long string that exceeds limit", fieldName: "Test", maxLength: 10).isValid == false)

*// Number validation tests*

let (numberResult, \_) = validateNumber("123.45", fieldName: "Test")

results["valid\_number"] = numberResult.isValid

let (invalidNumberResult, \_) = validateNumber("abc", fieldName: "Test")

results["invalid\_number"] = (invalidNumberResult.isValid == false)

*// Currency validation tests*

let (currencyResult, \_) = validateCurrency("$123.45", fieldName: "Test")

results["valid\_currency"] = currencyResult.isValid

let (invalidCurrencyResult, \_) = validateCurrency("-$50", fieldName: "Test")

results["negative\_currency"] = (invalidCurrencyResult.isValid == false)

*// Email validation tests*

results["valid\_email"] = validateEmail("test@example.com").isValid

results["invalid\_email"] = (validateEmail("invalid.email").isValid == false)

*// Date validation tests*

let (dateResult, \_) = validateDate("2024-01-01", fieldName: "Test")

results["valid\_date"] = dateResult.isValid

let (invalidDateResult, \_) = validateDate("invalid-date", fieldName: "Test")

results["invalid\_date"] = (invalidDateResult.isValid == false)

return results

}

*/// Performance test for validation functions*

public static func performanceTest(iterations: Int = 1000) -> TimeInterval {

let startTime = Date()

for \_ in 0..<iterations {

\_ = validateNotEmpty("Test String", fieldName: "Test")

let (\_, \_) = validateNumber("123.45", fieldName: "Test")

\_ = validateEmail("test@example.com")

let (\_, \_) = validateDate("2024-01-01", fieldName: "Test")

}

return Date().timeIntervalSince(startTime)

}

*/// Generate test validation errors for UI testing*

public static func sampleValidationErrors() -> [ValidationError] {

return [

.empty(field: "Username"),

.tooShort(field: "Password", minimum: 8, actual: 4),

.invalidEmail(email: "invalid.email"),

.budgetExceeded(category: "Groceries", amount: 150.0, limit: 100.0),

.invalidCurrency(amount: "invalid"),

.custom(message: "Custom validation error")

]

}

}

}

#endif

## Usage Examples:

swift

*// Example usage in ViewModels*

class PurchaseEntryViewModel: ObservableObject {

@Validated(validator: CommonValidationRules.budgetAmount.validate)

var amountText: String = ""

@Validated(validator: { ValidationHelpers.validateCategory($0, existingCategories: availableCategories) })

var categoryName: String = ""

var isFormValid: Bool {

return $amountText.isValid && $categoryName.isValid

}

}

*// Example usage in SwiftUI views*

struct PurchaseEntryView: View {

@StateObject private var viewModel = PurchaseEntryViewModel()

var body: some View {

Form {

TextField("Amount", text: $viewModel.amountText)

.validation($viewModel.$amountText, show: true)

TextField("Category", text: $viewModel.categoryName)

.validation($viewModel.$categoryName, show: true)

}

}

}

**Integration Points:**

* Used throughout the app for input validation and data integrity
* Provides consistent error messages and recovery suggestions
* Integrates with SwiftUI for real-time validation feedback
* Supports complex validation scenarios with rule building
* Coordinates with AppConstants for validation limits and rules
* Works with ErrorHandler for comprehensive error management

**UI Components (Utils/UI/)**

**58. Utils/UI/CommonComponents.swift**

**File Type:** Comprehensive UI Component Library  
**Purpose:** Reusable components with consistent styling, accessibility, and advanced functionality

**Core Component Architecture:**

swift

public struct CommonComponents {

*// Card-based layouts*

public struct CardView<Content: View>: View

public struct StatsCard: View

*// List components*

public struct ListRow<Leading: View, Trailing: View>: View

public struct TransactionRow: View

*// Input components*

public struct ValidatedTextField: View

public struct CurrencyTextField: View

public struct SearchablePicker<T: Hashable & CustomStringConvertible>: View

public struct CategoryPicker: View

*// Button components*

public struct PrimaryButton: View

public struct SecondaryButton: View

public struct FloatingActionButton: View

*// Display components*

public struct CategoryIcon: View

public struct ProgressBar: View

public struct BudgetStatusIndicator: View

*// Navigation components*

public struct CustomNavigationBar<Leading: View, Trailing: View>: View

public struct TabBarItem: View

*// Alert components*

public struct AlertView: View

}

**Advanced Card Components:**

swift

public struct StatsCard: View {

let title: String, value: String, subtitle: String?

let icon: String?, color: Color, trend: Trend?

public enum Trend {

case up(String), down(String), neutral(String)

var color: Color, icon: String, text: String

}

}

**Input Components with Validation:**

swift

public struct ValidatedTextField: View {

let title: String, placeholder: String

@Binding var text: String

let validation: (String) -> ValidationResult

let keyboardType: UIKeyboardType

let isSecure: Bool

public enum ValidationResult: Equatable {

case valid, invalid(String)

var isValid: Bool, errorMessage: String?

}

}

public struct CurrencyTextField: View {

let title: String

@Binding var amount: Double

let currencyCode: String, isRequired: Bool

private func validateCurrency(\_ input: String) -> ValidatedTextField.ValidationResult

}

**Advanced Selection Components:**

swift

public struct SearchablePicker<T: Hashable & CustomStringConvertible>: View {

let title: String, items: [T]

@Binding var selection: T?

let allowsCustomEntry: Bool

@State private var isPresented = false

@State private var searchText = ""

}

**Interactive Display Components:**

swift

public struct ProgressBar: View {

let progress: Double, total: Double, color: Color

let height: CGFloat, showPercentage: Bool

@State private var animatedProgress: Double = 0

}

public struct BudgetStatusIndicator: View {

let spent: Double, budget: Double, category: String

private var (statusText, statusColor): (String, Color)

}

**Alert and Feedback System:**

swift

public struct AlertView: View {

let title: String, message: String, alertType: AlertType

let primaryButton: AlertButton, secondaryButton: AlertButton?

public enum AlertType {

case info, warning, error, success

var color: Color, icon: String

}

public struct AlertButton {

let title: String, style: Style, action: () -> Void

public enum Style {

case `default`, destructive, cancel

}

}

}

**Pre-defined Component Helpers:**

swift

extension CommonComponents {

public static func quickAddButton(onTap: @escaping () -> Void) -> some View

public static func budgetOverviewCard(budgeted: Double, spent: Double, remaining: Double) -> some View

public static func emptyState(icon: String, title: String, message: String, actionTitle: String?, action: (() -> Void)?) -> some View

}

**View Extensions for Components:**

swift

extension View {

public func cardStyle(padding: EdgeInsets, cornerRadius: CGFloat, shadowRadius: CGFloat, backgroundColor: Color) -> some View

public func listRowStyle() -> some View

}

**Testing Support:**

swift

#if DEBUG

extension CommonComponents {

public struct TestComponents: View {

*// Complete preview environment for all components*

*// Multiple test scenarios and configurations*

}

}

#endif

**Integration Points:**

* Used throughout the app for consistent UI patterns
* Integrates with ThemeManager for consistent styling
* Provides accessibility support out of the box
* Supports complex interactions and validation

**59. Utils/UI/ErrorViews.swift**

**File Type:** Comprehensive Error Display System  
**Purpose:** Sophisticated error presentation with recovery options and accessibility

**Error View Architecture:**

swift

public struct ErrorViews {

*// Basic error display*

public struct BasicErrorView: View

*// Full-screen error handling*

public struct FullScreenErrorView: View

*// Inline error display*

public struct InlineErrorBanner: View

*// Specialized error views*

public struct NetworkErrorView: View

public struct EmptyStateView: View

public struct PermissionErrorView: View

*// Temporary notifications*

public struct ErrorToast: View

}

**Error Categorization and Display:**

swift

public struct BasicErrorView: View {

let error: AppError

let onDismiss: (() -> Void)?

let onRetry: (() -> Void)?

private var buttonSection: some View {

*// Retry and dismiss buttons based on error type*

}

}

public struct FullScreenErrorView: View {

let error: AppError

let onRetry: (() -> Void)?

let onSupport: (() -> Void)?

private var troubleshootingSection: some View {

*// Step-by-step troubleshooting for critical errors*

}

}

**Specialized Error Handling:**

swift

public struct NetworkErrorView: View {

let onRetry: (() -> Void)?

let onSettings: (() -> Void)?

@State private var isCheckingConnection = false

private var troubleshootingTips: some View {

*// Network-specific troubleshooting*

}

}

public struct PermissionErrorView: View {

let permissionType: AppError.PermissionType

let onOpenSettings: (() -> Void)?

let onDismiss: (() -> Void)?

public enum PermissionType {

case notifications, fileAccess, camera, photos

var description: String

var recoverySuggestion: String

}

private var permissionInstructions: some View {

*// Step-by-step permission enabling instructions*

}

}

**Interactive Error Toast:**

swift

public struct ErrorToast: View {

let error: AppError

let onDismiss: (() -> Void)?

@State private var isVisible = false

@State private var dragOffset: CGSize = .zero

*// Swipe-to-dismiss functionality*

*// Auto-dismiss after timeout*

*// Haptic feedback integration*

}

**Error View Modifiers:**

swift

public struct ErrorHandling: ViewModifier {

let context: String

let showInline: Bool

let onRetry: (() -> Void)?

@StateObject private var errorHandler = ErrorHandler.shared

@State private var showingErrorSheet = false

}

extension View {

public func errorHandling(context: String, showInline: Bool = true, onRetry: (() -> Void)? = nil) -> some View

public func errorToast(error: Binding<AppError?>, onDismiss: (() -> Void)? = nil) -> some View

}

**Pre-defined Error Views:**

swift

extension ErrorViews {

public static func networkError(onRetry: @escaping () -> Void) -> some View

public static func noDataAvailable(title: String = "No Data Available", message: String, actionTitle: String?, onAction: (() -> Void)?) -> some View

public static func budgetDataError(onRetry: @escaping () -> Void) -> some View

public static func importExportError(\_ error: AppError, onRetry: @escaping () -> Void) -> some View

}

**Supporting Components:**

swift

private struct TroubleshootingRow: View {

let icon: String, text: String

}

private struct InstructionRow: View {

let step: String, text: String

}

**Testing Support:**

swift

#if DEBUG

extension ErrorViews {

public struct TestErrorViews: View {

*// Complete error view testing interface*

*// Multiple error scenarios and configurations*

}

}

#endif

**Integration Points:**

* Integrates with ErrorHandler.shared for centralized error management
* Supports all AppError types with appropriate UI
* Provides accessibility support and user guidance
* Coordinates with system settings and recovery actions

**60. Utils/UI/LoadingViews.swift**

**File Type:** Advanced Loading State System  
**Purpose:** Sophisticated loading indicators with animations and progress tracking

**Loading View Architecture:**

swift

public struct LoadingViews {

*// Basic loading indicators*

public struct BasicLoadingView: View

*// Full-screen loading overlays*

public struct FullScreenLoadingView: View

*// Skeleton loading placeholders*

public struct SkeletonView: View

public struct ListLoadingView: View

public struct ListItemSkeleton: View

public struct CardLoadingView: View

*// Chart loading placeholders*

public struct ChartLoadingView: View

}

**Animated Loading Components:**

swift

public struct BasicLoadingView: View {

let message: String, showProgress: Bool

@State private var rotation: Double = 0

*// Rotating indicator animation*

}

public struct FullScreenLoadingView: View {

let title: String, subtitle: String?

let progress: Double?, showCancel: Bool

let onCancel: (() -> Void)?

@State private var animationOffset: CGFloat = 0

@State private var pulseOpacity: Double = 0.3

private var loadingIndicator: some View {

*// Pulsing and spinning animations*

}

private func progressSection(\_ progress: Double) -> some View {

*// Progress bar with percentage display*

}

}

**Skeleton Loading System:**

swift

public struct SkeletonView: View {

let height: CGFloat, cornerRadius: CGFloat

@State private var shimmerOffset: CGFloat = -200

*// Shimmer animation overlay*

*// Linear gradient animation*

*// 1.5-second repeat cycle*

}

public struct ListLoadingView: View {

let itemCount: Int, itemHeight: CGFloat

*// Staggered animation delays (0.1s per item)*

*// LazyVStack for performance*

}

public struct ListItemSkeleton: View {

let height: CGFloat

*// Icon placeholder (40x40 circle)*

*// Title and subtitle placeholders*

*// Trailing content placeholder*

}

**Chart Loading Placeholders:**

swift

public struct ChartLoadingView: View {

let chartType: ChartType

@State private var animationPhase: CGFloat = 0

public enum ChartType {

case line, bar, pie

}

@ViewBuilder

private var chartPlaceholder: some View {

switch chartType {

case .line: lineChartPlaceholder

case .bar: barChartPlaceholder

case .pie: pieChartPlaceholder

}

}

private var lineChartPlaceholder: some View {

*// Grid lines background*

*// Animated line path*

*// Data point indicators*

}

}

**Animation Specifications:**

* **Shimmer Duration:** 1.5 seconds linear repeat
* **Pulse Animation:** 1.0 second ease-in-out
* **Staggered Delays:** 0.1 seconds between list items
* **Rotation Animation:** Continuous 360° rotation
* **Chart Animation:** 0.8 second chart-specific animations

**Integration Points:**

* Used throughout the app for consistent loading states
* Provides smooth transitions between loading and content states
* Supports various loading scenarios (network, computation, file operations)
* Integrates with performance monitoring for loading time tracking

**📱 Widget System (Widget/)**

**61. Widget/Info.plist**

**File Type:** Widget Extension Configuration  
**Purpose:** Widget metadata and app group configuration

**Key Configuration:**

xml

<key>CFBundleDisplayName</key>

<string>Budget Widget</string>

<key>NSExtension</key>

<dict>

<key>NSExtensionPointIdentifier</key>

<string>com.apple.widgetkit-extension</string>

</dict>

<key>LSApplicationCategoryType</key>

<string>public.app-category.finance</string>

**Integration Points:**

* Configures widget as WidgetKit extension
* Sets display name and category for App Store
* Enables finance category classification

**62. Widget/Budget WidgetExtension.entitlements**

**File Type:** Widget Entitlements  
**Purpose:** App group access for data sharing

**Key Entitlements:**

xml

<key>com.apple.security.application-groups</key>

<array>

<string>group.com.brandontitensor.BrandonsBudget</string>

</array>

**Integration Points:**

* Enables shared data access with main app
* Required for widget data synchronization
* Matches main app entitlements for consistency

**63. Widget/Budget\_Widget.swift**

**File Type:** Complete Widget Implementation  
**Purpose:** Comprehensive widget system with multiple sizes, error handling, and rich data display

**Widget Architecture:**

swift

@main

struct BudgetWidget: Widget {

private let kind = "BudgetWidget"

var body: some WidgetConfiguration {

StaticConfiguration(

kind: kind,

provider: BudgetWidgetProvider()

) { entry in

BudgetWidgetView(entry: entry)

.containerBackground(.fill.tertiary, for: .widget)

}

.configurationDisplayName("Budget Status")

.description("Keep track of your monthly budget and spending at a glance.")

.supportedFamilies([.systemSmall, .systemMedium, .systemLarge])

}

}

**Widget Entry System:**

swift

struct BudgetWidgetEntry: TimelineEntry {

let date: Date

let budgetSummary: SharedDataManager.BudgetSummary?

let recentTransactions: [SharedDataManager.RecentTransaction]

let topCategories: [SharedDataManager.CategorySpending]

let widgetData: SharedDataManager.WidgetData?

let errorState: WidgetErrorState?

let lastUpdateDate: Date?

*// Convenience properties*

var isOverBudget: Bool

var percentageUsed: Double

var statusColor: Color

var displayTitle: String

var hasValidData: Bool

enum WidgetErrorState {

case noData, staleData, appGroupUnavailable, corruptedData

var displayTitle: String, displayMessage: String

var systemImage: String, color: Color

}

}

**Widget Provider with Performance Monitoring:**

swift

struct BudgetWidgetProvider: TimelineProvider {

private let sharedDataManager = SharedDataManager.shared

private let performanceMonitor = PerformanceMonitor.shared

func placeholder(in context: Context) -> BudgetWidgetEntry

func getSnapshot(in context: Context, completion: @escaping (BudgetWidgetEntry) -> Void)

func getTimeline(in context: Context, completion: @escaping (Timeline<BudgetWidgetEntry>) -> Void)

private func createEntry() -> BudgetWidgetEntry {

*// Data health checking*

*// Widget data retrieval*

*// Error state determination*

*// Performance monitoring*

}

private func calculateNextUpdateTime(from currentDate: Date, entry: BudgetWidgetEntry) -> Date {

*// Error state: 15 minutes*

*// Active hours (7 AM - 10 PM): 30 minutes*

*// Off hours: 2 hours*

}

}

**Multi-Size Widget Views:**

swift

struct BudgetWidgetView: View {

let entry: BudgetWidgetEntry

@Environment(\.widgetFamily) private var widgetFamily

var body: some View {

Group {

if entry.hasValidData {

switch widgetFamily {

case .systemSmall: SmallBudgetWidget(entry: entry)

case .systemMedium: MediumBudgetWidget(entry: entry)

case .systemLarge: LargeBudgetWidget(entry: entry)

default: SmallBudgetWidget(entry: entry)

}

} else {

ErrorBudgetWidget(entry: entry)

}

}

.widgetURL(URL(string: "brandonsbudget://widget"))

}

}

**Small Widget (Compact Display):**

swift

struct SmallBudgetWidget: View {

let entry: BudgetWidgetEntry

var body: some View {

VStack(spacing: 8) {

*// Header with status icon*

HStack {

Text("Budget")

.font(.caption)

.foregroundColor(.secondary)

Spacer()

Image(systemName: statusIcon)

.foregroundColor(entry.statusColor)

}

Spacer()

*// Main content - remaining budget*

VStack(spacing: 4) {

Text(budgetSummary.formattedRemainingBudget())

.font(.title2)

.fontWeight(.bold)

.foregroundColor(entry.statusColor)

Text(budgetSummThis documentation now serves as the \*\*definitive reference\*\* for understanding, debugging, and extending the Brandon's Budget application. It provides everything needed to work confidently with this codebase, identify integration issues, and maintain architectural consistency across all features.

---

## 🎨 Utils System (Utils/)

### Extensions (Utils/Extensions/)

#### 49. `Utils/Extensions/DateExtensions.swift`

\*\*File Type:\*\* Date Utility Extensions

\*\*Purpose:\*\* Comprehensive date manipulation with thread safety and robust error handling

\*\*Architecture:\*\*

```swift

public extension Date {

*// Thread-safe calendar instance*

private static let calendar: Calendar = {

var calendar = Calendar(identifier: .gregorian)

calendar.timeZone = .current

return calendar

}()

}

**Core Calendar Operations:**

swift

*// Boundary calculations*

var startOfDay: Date *// Beginning of day (00:00:00)*

var endOfDay: Date *// End of day (23:59:59)*

var startOfMonth: Date *// First day of month*

var endOfMonth: Date *// Last day of month*

var startOfYear: Date *// January 1st*

var endOfYear: Date *// December 31st*

*// Component access*

var month: Int *// 1-12*

var year: Int *// Full year*

var day: Int *// Day of month*

var weekday: Int *// 1-7 (Sunday = 1)*

**Date Checks and Validation:**

swift

var isToday: Bool *// Is today's date*

var isYesterday: Bool *// Is yesterday's date*

var isTomorrow: Bool *// Is tomorrow's date*

var isInCurrentMonth: Bool *// Same month/year as now*

var isInCurrentYear: Bool *// Same year as now*

var isPast: Bool *// Before current time*

var isFuture: Bool *// After current time*

**Advanced Date Manipulation:**

swift

func adding(days: Int) -> Date *// Add/subtract days*

func adding(months: Int) -> Date *// Add/subtract months*

func adding(years: Int) -> Date *// Add/subtract years*

func daysBetween(\_ date: Date) -> Int *// Days difference*

func monthsBetween(\_ date: Date) -> Int *// Months difference*

func yearsBetween(\_ date: Date) -> Int *// Years difference*

**Formatting Utilities:**

swift

var relativeDescription: String *// "2 days ago"*

var monthYearString: String *// "January 2024"*

func formatted(style: DateFormatter.Style = .medium) -> String

*// Specialized formatters*

static func from(year: Int, month: Int, day: Int) -> Date?

static func fromISO8601(\_ string: String) -> Date?

**Testing Support:**

swift

#if DEBUG

static func testDate(year: Int, month: Int, day: Int) -> Date

#endif

**Integration Points:**

* Used throughout app for consistent date handling
* Thread-safe implementation prevents calendar conflicts
* Provides localized formatting with timezone awareness
* Supports testing with mock date generation

**🎯 Complete Application Architecture Summary**

**Final File Count: 75 Total Files**

**Application Structure:**

* **5 App Configuration Files:** Core app setup and entitlements
* **5 Core Models:** Data structures with validation and business logic
* **4 Core Services:** Data persistence, import/export, error handling
* **4 Core Managers:** Budget, settings, notifications, theme management
* **3 Core Types:** Enums, time periods, shared data management
* **16 Feature Views:** Complete UI implementation across all features
* **7 Feature ViewModels:** Business logic and state management
* **12 Reusable UI Components:** Shared components and specialized interfaces
* **4 Utility Extensions:** Date, number, color, and view enhancements
* **2 Utility Constants:** App configuration and color definitions
* **3 Utility Helpers:** Formatting, performance monitoring, validation
* **3 Utility UI Components:** Common components, error views, loading states
* **3 Widget Files:** Complete widget implementation with multi-size support
* **4 Additional Configuration Files:** Widget setup and entitlements

**Architecture Excellence:**

**Design Patterns:**

* **MVVM Architecture:** Consistent ViewModel pattern across all features
* **Centralized Management:** All core functionality managed through singleton services
* **Environment Object Injection:** Proper dependency injection throughout UI hierarchy
* **Publisher-Subscriber:** Reactive programming with Combine for data flow
* **Error-First Design:** Comprehensive error handling with recovery mechanisms

**Performance Optimization:**

* **Async/Await:** All heavy operations use modern concurrency
* **Background Processing:** Data operations performed off main thread
* **Caching Systems:** Strategic caching with TTL for frequently accessed data
* **Performance Monitoring:** Real-time performance tracking and optimization
* **Memory Management:** Proper resource cleanup and memory pressure handling

**Integration Excellence:**

* **Loose Coupling:** Features communicate through managers and NotificationCenter
* **Type Safety:** Comprehensive use of Swift's type system for reliability
* **Accessibility:** Complete accessibility implementation across all components
* **Localization:** Full localization support with proper string management
* **Testing Support:** Comprehensive testing utilities and mock data generation

**User Experience:**

* **Consistent Theming:** Centralized color and styling management
* **Error Recovery:** User-friendly error messages with actionable recovery steps
* **Loading States:** Sophisticated loading indicators and skeleton screens
* **Animation:** Smooth, purposeful animations throughout the interface
* **Widget Integration:** Rich widget experience with comprehensive data display

**Critical Integration Points:**

**Data Flow Architecture:**

User Action → Feature View → ViewModel → Manager → Service → Core Data → Widget Update

↓ ↓ ↓ ↓ ↓ ↓ ↓

UI Update ← Published ← ObservableObject ← Publisher ← Change ← Persistence ← Notification

**Error Handling Flow:**

Error Source → AppError Transformation → ErrorHandler.shared → UI Presentation

↓ ↓ ↓ ↓

Context Logging → Error Categorization → Recovery Options → User Action

**Cross-Feature Communication:**

Feature A → NotificationCenter.post() → Feature B Reception → Action Execution

↓ ↓ ↓ ↓

Context Data → Custom Notification → Target Navigation → State Update

**Debugging and Maintenance Excellence:**

This comprehensive documentation provides:

* **Complete architectural understanding** for all 75 files
* **Detailed integration patterns** showing how components work together
* **Extensive debugging checklists** for common issues and integration problems
* **Performance monitoring guidance** for optimization and maintenance
* **Accessibility validation** ensuring inclusive user experience
* **Testing support** with mock data and validation utilities

The Brandon's Budget application represents a sophisticated, well-architected iOS application with enterprise-level code organization, comprehensive error handling, and excellent user experience design. This documentation serves as the definitive reference for understanding, debugging, extending, and maintaining the entire codebase.This documentation now serves as the **definitive reference** for understanding, debugging, and extending the Brandon's Budget application. It provides everything needed to work confidently with this codebase, identify integration issues, and maintain architectural consistency across all features.