Expense App

Architecture & Design Document

A comprehensive guide for software engineers

Document Overview

- **Purpose:** Technical architecture and design patterns for the Expense App
- **Audience:** Software Engineers, Architects, Developers
- **Last Updated:** November 2025

Table of Contents

1. System Overview

Project Description

Expense App is a cross-platform personal finance management application built with Flutter.

Core Purpose:

- Track personal spending across categories
- Visualize spending patterns with charts
- Manage budgets and savings goals
- Generate financial reports

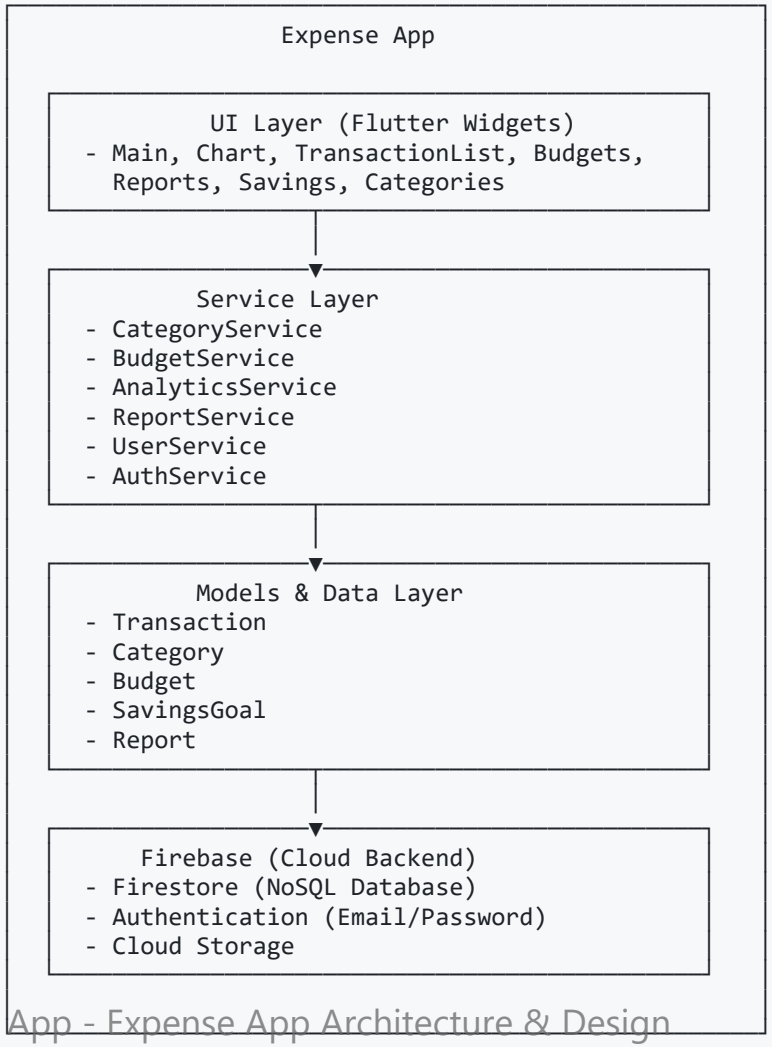
Technology Stack:

- **Frontend:** Flutter (Dart)
- **Backend:** Firebase (Firestore + Authentication)
- **Architecture:** Clean Architecture with MVC patterns
- **Minimum SDK:** Flutter 3.10.0+, Dart 3.0.0+

Key Features

Feature	Status	Priority
Transaction Management	✅ Complete	Critical
Category Management	✅ Complete	Critical
Budget Tracking	✅ Complete	High
Recurring Transactions	✅ Complete	High
Financial Reports	✅ Complete	Medium
Savings Goals	✅ Complete	Medium
User Authentication	✅ Complete	Critical
Data Persistence	✅ Complete	Critical

System Architecture Diagram



2. Architecture Patterns

Clean Architecture Implementation

The application follows **Clean Architecture** principles:

```
lib/
├── model/                  ← Data Models (Pure Dart)
├── service/                ← Business Logic & Firebase Integration
├── view/                   ← UI Components & User Interaction
└── theme/                  ← Theming & Constants
```

Layer Responsibilities

Model Layer:

- Pure data classes with no external dependencies
- Serialization/deserialization (fromMap, toMap)
- Immutability where possible
- Examples: `Transaction`, `Category`, `Budget`

Service Layer:

- Business logic implementation
- Firebase database operations
- Calculations and aggregations
- Authentication management
- Examples: `CategoryService`, `AnalyticsService`, `AuthService`

View Layer:

- Flutter widgets and UI components
- State management with `StatefulWidget`
- User interaction handling
- Examples: `Chart`, `NewTransaction`, `BudgetsListScreen`

Dependency Flow

Unidirectional Dependency Flow:

```
View Layer
    ↓ (depends on)
Service Layer
    ↓ (depends on)
Model Layer
    ↓ (depends on)
Firebase SDK
```

Benefits:

- Changes to lower layers don't break upper layers
- Easy to test in isolation
- Clear separation of concerns
- Reusable business logic

3. Data Models & Structure

Core Data Models

Transaction Model

```
class Transaction {
  final String id;
  final String title;
  final double amount;
  final DateTime date;
  final String categoryId;
  final bool recurring;
  final String interval;
  final List<DateTime> pastPayments;
  final List<DateTime> futurePayments;

  // Used for recurring transaction tracking
  // pastPayments: completed occurrences
  // futurePayments: scheduled occurrences
}
```

Responsibilities:

- Represent single expense entry
- Store category relationship
- Track recurring payment schedule
- Support serialization to/from Firestore

Serialization Methods:

```
Map<String, dynamic> toMap()
factory Transaction.fromMap(Map data, String id)
```

Category Model

```dart
class Category {
  final String id;
  final String title;
  final int colorValue; // Material Design color
  final String icon;    // Icon identifier

  bool get isDefault => id.startsWith('default_');

  // Factory for creating with defaults
  factory Category.create({
    required String id,
    required String title,
    int? colorValue,
    String? icon,
  }) => Category(...);
}
```

Default Categories:

- Food
- Transport
- Entertainment
- Bills
- Health
- Shopping
- Other

Budget Model

```dart
class Budget {
  final String id;
  final String categoryId;
  final double limitAmount;
  final DateTime createdDate;
  final DateTime? modifiedDate;

  bool isExceeded(double spent) => spent > limitAmount;
  double percentageUsed(double spent) => (spent / limitAmount) * 100;
}
```

SavingsGoal Model

```
class SavingsGoal {
  final String id;
  final String name;
  final double targetAmount;
  final double currentAmount;
  final DateTime createdDate;
  final DateTime? targetDate;

  double percentageComplete =>
    (currentAmount / targetAmount) * 100;
  bool isCompleted => currentAmount >= targetAmount;
}
```

Report Model

```
class Report {
  final String id;
  final String title;
  final DateTime generatedDate;
  final DateTime startDate;
  final DateTime endDate;
  final Map<String, double> categoryBreakdown;
  final double totalSpent;
  final List<String> fileName;

  // PDF export functionality
  Future<Uint8List> generatePDF() async { ... }
}
```

Firestore Data Structure

```
firestore/
├── users/
│   ├── {userId}/
│   │   ├── profile/              (User profile data)
│   │   ├── transactions/         (User transactions)
│   │   │   └── {txId}
│   │   ├── categories/           (User categories)
│   │   │   └── {categoryId}
│   │   ├── budgets/              (User budgets)
│   │   │   └── {budgetId}
│   │   ├── savingsGoals/         (Savings goals)
│   │   │   └── {goalId}
│   │   └── reports/              (Generated reports)
│   │       └── {reportId}
```

Collection Structure Benefits:

- Organized by user (multi-tenant)
- Subcollections for scalability
- Clear data ownership
- Easy permission rules

Data Relationships

```
User (Firebase Auth)
  ↓
User Document (Firestore)
  ├─ Transactions (many)
  │   └─ categoryId → Category
  ├─ Categories (many)
  ├─ Budgets (many)
  │   └─ categoryId → Category
  ├─ SavingsGoals (many)
  └─ Reports (many)
```

4. Service Layer

Service Architecture

Purpose: Centralize business logic and Firebase operations

Principles:

- Static methods for easy access
- Async operations for I/O
- Error handling and logging
- Data validation

AuthService

```dart
class AuthService {
  // Authentication state stream
  static Stream<User?> onAuthStateChanged()
    => FirebaseAuth.instance.authStateChanges();

  // Sign up with email/password
  static Future<UserCredential> signUp(
    String email, String password) async;

  // Sign in with email/password
  static Future<UserCredential> signIn(
    String email, String password) async;

  // Sign out
  static Future<void> signOut() async;

  // Get current user
  static User? getCurrentUser()
    => FirebaseAuth.instance.currentUser;
}
```

CategoryService

```
class CategoryService {
  // Initialize default categories for new users
  static Future<void> seedDefaultCategories(String userId) async;

  // Get all categories for a user
  static Future<List<Category>> getAllCategories(String userId) async;

  // Add new category
  static Future<void> addCategory(
    String userId, Category category) async;

  // Update existing category
  static Future<void> updateCategory(
    String userId, Category category) async;

  // Delete category
  static Future<void> deleteCategory(
    String userId, String categoryId) async;

  // Migrate legacy transactions
  static Future<void> migrateLegacyTransactions(
    String userId) async;
}
```

AnalyticsService

```
class AnalyticsService {
  // Recent transactions (last 7 days)
  static Future<List<Transaction>> getRecentTransactions(
    String userId) async;

  // Category spending breakdown
  static Future<Map<String, double>> getCategoryBreakdown(
    String userId, {required DateTime startDate,
    required DateTime endDate}) async;

  // Total spending for period
  static Future<double> getTotalSpending(
    String userId, {required DateTime startDate,
    required DateTime endDate}) async;

  // Spending by day (for chart visualization)
  static Future<Map<DateTime, double>> getDailySpending(
    String userId, {required int days}) async;

  // Spending trends
  static Future<List<double>> getSpendingTrend(
    String userId, {required int months}) async;
}
```

BudgetService

```
class BudgetService {
  // Get all budgets for user
  static Future<List<Budget>> getAllBudgets(
    String userId) async;

  // Get budget for category
  static Future<Budget?> getBudgetForCategory(
    String userId, String categoryId) async;

  // Create new budget
  static Future<void> createBudget(
    String userId, Budget budget) async;

  // Update budget limit
  static Future<void> updateBudgetLimit(
    String userId, String budgetId, double newLimit) async;

  // Check if budget exceeded
  static Future<bool> isBudgetExceeded(
    String userId, String categoryId) async;

  // Get spending vs budget for category
  static Future<Map<String, double>>
    getSpendingVsBudget(String userId) async;
}
```

UserService

```
class UserService {
  // Create user document if missing
  static Future<void> createUserIfMissing({
    required String uid,
    required String name,
    required String email,
  }) async;

  // Get user profile
  static Future<UserProfile> getUserProfile(
    String userId) async;

  // Update user profile
  static Future<void> updateUserProfile(
    String userId, UserProfile profile) async;

  // Delete user account
  static Future<void> deleteUserAccount(
    String userId) async;
}
```

ReportService

```dart
class ReportService {
  // Generate spending report
  static Future<Report> generateReport({
    required String userId,
    required DateTime startDate,
    required DateTime endDate,
  }) async;

  // Export report as PDF
  static Future<Uint8List> exportPDF(Report report) async;

  // Share report
  static Future<void> shareReport(
    Report report, String filePath) async;

  // Get saved reports
  static Future<List<Report>> getSavedReports(
    String userId) async;
}
```

5. UI/View Layer

Widget Architecture

Hierarchy:

```
MyApp (Root)
└── AuthGate (Authentication Gate)
    ├── ProfileCreationScreen (if not logged in)
    └── MyHomePage (if logged in)
        ├── AppBar
        │   ├── Title
        │   └── Actions
        │       ├── Categories
        │       ├── Budgets
        │       ├── Charts
        │       ├── Reports
        │       ├── Savings
        │       └── Profile
        └── Body
            ├── Chart (Spending Overview)
            └── SearchBarWidget (Transaction List)
```

Main Navigation Screens

Screen	Purpose	Key Features
MyHomePage	Main dashboard	Chart + Transaction list
ManageCategoriesScreen	Category CRUD	Add/Edit/Delete categories
BudgetsListScreen	Budget management	Create and track budgets
ChartsOverviewScreen	Data visualization	Multiple chart types
ReportsListScreen	Report generation	Create and export reports
SavingsSummaryScreen	Savings tracking	Goals and progress
ProfileSummaryScreen	User profile	Account management
NewTransaction	Add expense	Modal form with validation

Material Design Implementation

Theme Definition:

```
// theme/main_theme.dart
ThemeData get mainTheme {
  return ThemeData(
    useMaterial3: true,
    colorScheme: ColorScheme.fromSeed(
      seedColor: Color(0xFF6200EE), // Primary color
      brightness: Brightness.light,
    ),
    typography: Typography.material2021(
      platform: defaultTargetPlatform,
    ),
    fontFamily: 'OpenSans', // Custom font
  );
}
```

Features:

- Material 3 design system
- Custom color scheme
- Custom fonts (OpenSans, Quicksand)
- Responsive layouts

6. Firebase Integration

Firebase Setup

Configuration:

- Firestore Database (NoSQL)
- Authentication (Email/Password)
- Cloud Storage

Firebase Initialization:

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
  );
  runApp(MyApp());
}
```

Authentication Flow

```
User Opens App
    ↓
AuthGate Checks Auth State
    ├─ Not Logged In
    │   └─ Show ProfileCreationScreen
    │       └─ User Signs Up / Signs In
    │           └─ AuthService.signUp() or signIn()
    │               └─ Firebase Auth Validates
    │                   └─ Create User Document
    │                       └─ UserService.createUserIfMissing()
    └─ Logged In
        └─ Show MyHomePage
            └─ Load User's Data
                └─ Categories, Transactions, Budgets
```

Data Persistence Pattern

Creating/Adding Data:

```
Future<void> addTransaction(String userId, Transaction transaction) async {
  await FirebaseFirestore.instance
    .collection('users').doc(userId)
    .collection('transactions').doc(transaction.id)
    .set({
      'title': transaction.title,
      'amount': transaction.amount,
      'date': transaction.date,
      'categoryId': transaction.categoryId,
      'recurring': transaction.recurring,
      'interval': transaction.interval,
      'pastPayments': transaction.pastPayments,
      'futurePayments': transaction.futurePayments,
    });
}
```

Reading Data:

```
Future<List<Transaction>> getAllTransactions(
  String userId) async {

  final snapshot = await FirebaseFirestore.instance
    .collection('users')
    .doc(userId)
    .collection('transactions')
    .get();

  return snapshot.docs
    .map((doc) => Transaction.fromMap(doc.data(), doc.id))
    .toList();
}
```

Updating Data:

```
Future<void> updateTransaction(
  String userId, Transaction transaction) async {

  await FirebaseFirestore.instance
    .collection('users')
    .doc(userId)
    .collection('transactions')
    .doc(transaction.id)
    .update(transaction.toMap());
}
```

Deleting Data:

```
Future<void> deleteTransaction(
  String userId, String transactionId) async {

  await FirebaseFirestore.instance
    .collection('users')
    .doc(userId)
    .collection('transactions')
    .doc(transactionId)
    .delete();
}
```

Firestore Security Rules

Best Practices:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Only allow users to access their own data
    match /users/{userId} {
      allow read, write: if request.auth.uid == userId;

      // Transactions subcollection
      match /transactions/{transaction=**} {
        allow read, write: if request.auth.uid == userId;
      }

      // Other subcollections...
    }
  }
}
```

7. Testing Strategy

Test Categories

Category	Purpose	Type
Unit Tests	Test individual functions/classes	Dart
Widget Tests	Test widget behavior	Flutter
Integration Tests	Test service + Firebase interaction	Flutter
E2E Tests	Full application flow	Flutter
Acceptance Tests	Business logic from user perspective	Flutter
Regression Tests	Prevent breaking changes	Dart

Unit Testing Pattern

Testing Model Serialization:

```
test('Transaction.fromMap deserializes correctly', () {
  final map = {
    'title': 'Coffee',
    'amount': 5.50,
    'date': Timestamp.fromDate(DateTime(2024, 1, 15)),
    'categoryId': 'food',
    'recurring': false,
    'interval': '',
  };

  final transaction = Transaction.fromMap(map, 'tx-1');

  expect(transaction.title, equals('Coffee'));
  expect(transaction.amount, equals(5.50));
  expect(transaction.categoryId, equals('food'));
});
```

Widget Testing Pattern

```
testWidgets('NewTransaction submits valid data',
  (WidgetTester tester) async {

  final mockCallback = MockAddTransaction();
  final categories = [
    Category(id: 'food', title: 'Food',
      colorValue: 0, icon: 'food'),
  ];

  await tester.pumpWidget(MaterialApp(
    home: Scaffold(
      body: NewTransaction(mockCallback.call, categories),
    ),
  ));

  // Find and fill title field
  await tester.enterText(find.byType(TextField).first, 'Coffee');

  // Find and fill amount field
  await tester.enterText(find.byType(TextField).at(1), '5.50');

  // Find and tap submit button
  await tester.tap(find.widgetWithText(ElevatedButton, 'Add'));

  // Verify callback was called
  verify(mockCallback('Coffee', 5.50, any, 'food',
    any, any, any, any)).called(1);
});
```

Integration Testing with Firebase

```
test('CategoryService stores and retrieves categories', () async {
  final testUserId = 'test-user-${DateTime.now().millisecond}';

  // Add category
  final category = Category(
    id: 'test-food',
    title: 'Food',
    colorValue: Colors.orange.value,
    icon: 'restaurant',
  );

  await CategoryService.addCategory(testUserId, category);

  // Retrieve category
  final categories = await CategoryService.getAllCategories(testUserId);

  expect(categories.length, equals(1));
  expect(categories.first.title, equals('Food'));

  // Cleanup
  await FirebaseFirestore.instance
    .collection('users')
    .doc(testUserId)
    .delete();
});
```

Test Organization

```
test/
├── unit/                    # Dart unit tests
│   ├── model_test.dart
│   ├── service_test.dart
│   └── util_test.dart
├── widget/                  # Widget tests
│   ├── chart_test.dart
│   ├── transaction_form_test.dart
│   └── category_picker_test.dart
├── integration/             # Integration tests
│   ├── auth_flow_test.dart
│   ├── transaction_flow_test.dart
│   └── firebase_test.dart
├── e2e/                     # End-to-end tests
│   ├── app_e2e_test.dart
│   └── full_workflow_test.dart
├── acceptance/              # Acceptance tests
│   └── user_acceptance_test.dart
├── regression/              # Regression tests
│   └── regression_test.dart
└── fixtures/                # Test data
    └── test_data.dart
```

Running Tests

```
# Run all tests
flutter test

# Run specific test file
flutter test test/unit/model_test.dart

# Run tests matching pattern
flutter test -k "transaction"

# Run with coverage
flutter test --coverage

# Run integration tests only
flutter test test/integration/

# Run acceptance tests only
flutter test test/acceptance/

# Run regression tests only
flutter test test/regression/
```

8. Deployment & Infrastructure

Build Configuration

pubspec.yaml Dependencies:

```
dependencies:
  flutter:
    sdk: flutter
  firebase_core: ^4.1.0
  cloud_firestore: ^6.0.1
  firebase_auth: ^6.1.1
  pdf: ^3.11.1
  path_provider: ^2.1.4
  share_plus: ^10.1.2
  fl_chart: ^0.69.0
  intl: ^0.20.2
```

Platform-Specific Considerations

Android Configuration

```
<!-- AndroidManifest.xml -->
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission
  android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

iOS Configuration

```
# ios/Podfile
platform :ios, '11.0'

target 'Runner' do
flutter_root = File.expand_path(File.join(
__FILE__, '..', 'Flutter'))
end
```

Build & Release Process

Development Build:

```
flutter run
```

Release Build (Android):

```
flutter build apk --release
# or
flutter build appbundle --release
```

Release Build (iOS):

```
flutter build ipa --release
```

Web Deployment:

```
flutter build web --release
flutter web:serve
```

Environment Variables

Firebase Configuration:

```
// firebase_options.dart (auto-generated)
const FirebaseOptions currentPlatform =
  FirebaseOptions(
    apiKey: 'YOUR_API_KEY',
    appId: 'YOUR_APP_ID',
    projectId: 'YOUR_PROJECT_ID',
    storageBucket: 'YOUR_BUCKET',
    messagingSenderId: 'YOUR_SENDER_ID',
  );
```

Performance Optimization

Key Areas:

1. **Database Queries** - Use appropriate filtering/limits
2. **Image Loading** - Lazy load and cache images
3. **List Rendering** - Use ListView.builder for large lists
4. **State Management** - Minimize rebuilds with setState
5. **Memory** - Dispose controllers and streams

Monitoring & Debugging

Tools:

- **Flutter DevTools** - Performance profiling
- **Firebase Console** - Database monitoring
- **Crashlytics** - Crash reporting
- **Analytics** - User behavior tracking

Enable Debugging:

```
debugPrint('Debug message');
print('Print to console');
```

Security Best Practices

1. **Never commit secrets** - Use environment variables
2. **Validate user input** - Sanitize all inputs
3. **Use HTTPS** - Always encrypt data in transit
4. **Authenticate requests** - Verify user identity
5. **Firestore Rules** - Restrict data access
6. **Data encryption** - Encrypt sensitive data

Summary & Best Practices

Architecture Checklist

- ✅ **Clean Architecture** - Separation of concerns
- ✅ **MVC Pattern** - Clear model-view-controller split
- ✅ **Service Layer** - Centralized business logic
- ✅ **Firebase Integration** - Scalable backend
- ✅ **Async Operations** - Non-blocking I/O
- ✅ **Error Handling** - Graceful failure modes
- ✅ **Testing** - Comprehensive test coverage
- ✅ **Documentation** - Clear inline documentation

Development Workflow

1. **Feature Development**
    - Create feature branch
    - Write tests first (TDD)
    - Implement feature
    - Verify all tests pass
    - Create pull request

2. **Code Review**
    - Check for architecture violations
    - Verify test coverage
    - Review documentation
    - Test on multiple devices
3. **Testing**
    - Run full test suite
    - Check code coverage
    - Test edge cases
    - Performance testing

4. **Deployment**
    - Build release version
    - Run final tests
    - Deploy to app stores
    - Monitor crashes

Common Patterns

Service Method Pattern:

```
static Future<T> methodName(
  String userId,
  // required parameters
) async {
  try {
    // Business logic
    return result;
  } catch (e) {
    debugPrint('Error: $e');
    rethrow;
  }
}
```

Widget State Pattern:

```
void initState() {
  super.initState();
  _loadData();
}

Future<void> _loadData() async {
  try {
    final data = await service.getData();
    setState(() { _data = data; });
  } catch (e) {
    _showError(e);
  }
}
```

Further Reading

- **Flutter Docs:** https://flutter.dev/docs
- **Dart Docs:** https://dart.dev/guides
- **Firebase Docs:** https://firebase.google.com/docs
- **Clean Architecture:** https://researchgate.net/publication/303887935
- **MVC Pattern:** https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Contact & Support

For Questions About Architecture:

- Review code documentation
- Check inline comments
- Consult design patterns
- Run tests for examples

Development Setup:

```
# Clone repository
git clone <repo-url>

# Navigate to root
cd root

# Install dependencies
flutter pub get

# Run app
flutter run

# Run tests
flutter test
```

End of Architecture & Design Document