Standard Operating Procedure (SOP)

SplitFlow Frontend Development

Document Control

Project Name: SplitFlow (Investment Management Application)
Document Version: 1.0.3
Last Updated: February 9, 2026
Prepared By: Development Team
Document Status: Active
Classification: Internal Use Only

---

1. Executive Summary

1.1 Purpose

This Standard Operating Procedure (SOP) serves as the comprehensive guide for frontend development of the SplitFlow application. It establishes standardized protocols, workflows, and best practices to ensure consistency, quality, and efficiency throughout the development lifecycle.

1.2 Scope

This document covers all aspects of frontend development including environment setup, coding standards, testing procedures, deployment protocols, and maintenance guidelines. It is intended for use by developers, quality assurance personnel, and project stakeholders involved in the SplitFlow frontend development process.

1.3 Application Overview

SplitFlow is a collaborative investment management application designed to streamline project fund management, expense tracking, and member contribution oversight through a robust approval system. The application facilitates transparent financial operations among project stakeholders with role-based access controls and comprehensive audit trails.

Key Capabilities:

- Multi-user project fund management
- Real-time expense tracking and approval workflows
- Dynamic member role administration
- Contribution analytics and reporting
- Secure authentication and authorization

The frontend architecture leverages React Native (Expo) to deliver cross-platform compatibility with primary focus on Android deployment, ensuring optimal performance and user experience on mobile devices.

---

2. Technology Stack & Infrastructure

2.1 Core Technologies

2.1.1 React Native (Expo SDK 50+)

Description: React Native is a JavaScript framework for building native mobile applications. Expo is a platform built around React Native that provides additional tools and services for rapid development and deployment.

Key Features:

- Cross-platform compatibility (iOS and Android from single codebase)
- Hot reloading for immediate code change reflection
- Native component access through JavaScript APIs
- Extensive community support and ecosystem

Use Case in SplitFlow: Serves as the primary framework for building the mobile interface, enabling efficient development while maintaining native performance and user experience.

2.1.2 JavaScript (ES6+)

Description: Modern JavaScript with ECMAScript 6+ features providing enhanced syntax, module system, and functional programming capabilities.

Key Features:

- Arrow functions for concise syntax
- Template literals for string manipulation
- Destructuring for efficient data extraction
- Promises and async/await for asynchronous operations
- Classes for object-oriented programming

Use Case in SplitFlow: Powers all application logic, state management, and business rule implementation across the frontend.

2.1.3 Node.js

Description: A JavaScript runtime built on Chrome's V8 engine that executes JavaScript code outside of a web browser.

Key Features:

- Package management through npm (Node Package Manager)
- Server-side JavaScript execution
- Extensive module ecosystem
- Event-driven, non-blocking I/O model

Use Case in SplitFlow: Manages project dependencies, runs build scripts, and provides the runtime environment for development tools.

2.2 Development Environment & Tools

2.2.1 Visual Studio Code

Description: A lightweight yet powerful source code editor developed by Microsoft with extensive extension support.

Key Features:

- IntelliSense for intelligent code completion
- Integrated debugging capabilities
- Built-in Git integration
- Extensions marketplace for enhanced functionality
- Customizable workspace settings

Configuration for SplitFlow:

- Install ESLint extension for code quality
- Install Prettier extension for code formatting
- Install React Native Tools extension
- Install ES7+ React/Redux/React-Native snippets

2.2.2 Expo Go

Description: A mobile application available on iOS and Android that allows developers to open and test Expo projects on physical devices without building native applications.

Key Features:

- Instant preview of code changes
- No build process required for testing
- QR code scanning for quick access
- Over-the-air updates during development

Use Case in SplitFlow: Facilitates rigorous real-time testing on physical Android devices, enabling developers to validate functionality, performance, and user experience in actual device environments.

2.2.3 Android Studio

Description: The official Integrated Development Environment (IDE) for Android development, providing comprehensive tools for building, testing, and debugging Android applications.

Key Features:

- Android Virtual Device (AVD) Manager for emulation
- APK build and signing capabilities
- Performance profiling tools
- Layout inspector and debugging tools

Use Case in SplitFlow:

- Emulates various Android devices for testing compatibility
- Generates production-ready APK builds for deployment
- Provides debugging tools for platform-specific issues

2.2.4 Git & GitHub

Description: Git is a distributed version control system, while GitHub is a cloud-based hosting service for Git repositories.

Key Features:

- Complete change history tracking
- Branch management for parallel development
- Collaboration tools (pull requests, code review)
- Issue tracking and project management
- Backup and disaster recovery

Use Case in SplitFlow: Maintains complete version history of the codebase, enables collaborative development, and serves as the central repository for all project files.

2.2.5 Ngrok

Description: A secure tunneling service that creates public URLs for locally running applications, bypassing network restrictions.

Key Features:

- HTTPS tunneling for secure connections
- No configuration required on firewalls
- Temporary public URLs for local servers
- Request inspection and replay

Use Case in SplitFlow: Enables testing on physical devices across different networks by creating a secure tunnel to the local development server. Essential when developer and testing devices are on different Wi-Fi networks.

Command: `npm run tunnel`

2.3 Dependencies & Libraries

2.3.1 React Navigation

Description: A routing and navigation library for React Native applications, providing native-like navigation patterns.

Implementation in SplitFlow:

- Stack Navigator for screen transitions
- Hierarchical navigation structure
- Screen-to-screen parameter passing
- Navigation header customization

Key Screens:

- Authentication stack (Login, Registration)
- Main application stack (Dashboard, Project Details, Member Management)
- Modal overlays (Notifications, Settings)

2.3.2 @expo/vector-icons

Description: A comprehensive icon library providing thousands of customizable vector icons from popular icon sets.

Available Icon Sets:

- MaterialIcons
- FontAwesome
- Ionicons
- MaterialCommunityIcons

Use Case in SplitFlow: Provides consistent, scalable iconography throughout the application interface for buttons, navigation elements, status indicators, and visual cues.

2.3.3 Expo Linear Gradient

Description: A component that renders smooth color gradients, enhancing visual aesthetics of the user interface.

Use Case in SplitFlow: Creates visually appealing backgrounds for cards, headers, and UI elements, contributing to the modern and professional appearance of the application.

2.3.4 Additional Utilities

All dependencies are managed through `package.json` with specific version locking to ensure consistency across development environments. Regular dependency audits are performed to identify security vulnerabilities and outdated packages.

---

3. Project Architecture & Structure

3.1 Directory Organization

```
SplitFlowAndroid/
├── assets/                     # Static resources and media files
│   ├── images/                 # Application images and graphics
│   ├── icons/                  # Icon assets for various resolutions
│   └── splash/                 # Splash screen resources
│
├── src/                        # Source code directory
│   │
│   ├── components/             # Reusable UI components
│   │   ├── Theme.js            # Global theming constants (colors, fonts,
spacing)
│   │   ├── ProfileMenu.js      # User profile menu component
│   │   └── [other components]  # Additional shared components
│   │
│   ├── data/                   # Data layer and mock data
│   │   └── mockData.js         # Simulated backend data for development
│   │
│   ├── screens/                # Application screen components
│   │   │
│   │   ├── auth/               # Authentication related screens
│   │   │   ├── LoginScreen.js
│   │   │   └── RegisterScreen.js
│   │   │
│   │   ├── investor/           # Core application functionality
│   │   │   ├── DashboardScreen.js
│   │   │   ├── ProjectDetailScreen.js
│   │   │   ├── ManageInvestorsScreen.js
│   │   │   ├── AddSpendingScreen.js
│   │   │   └── [other screens]
│   │   │
│   │   └── notification/       # Notification center screens
│   │       └── NotificationScreen.js
│   │
│   ├── services/               # Service layer for API and external
interactions
│   │   ├── api.js              # API communication service
│   │   └── notificationService.js  # Push notification handling
│   │
│   └── utils/                  # Helper functions and utilities
│       ├── formatters.js       # Data formatting functions
│       ├── validators.js       # Input validation functions
│       └── [other utilities]
│
├── App.js                      # Application entry point and navigation
setup
```

```
├── app.json                    # Expo configuration (name, version, icons,
splash)
├── package.json                # Dependency management and npm scripts
└── babel.config.js             # Babel transpiler configuration
```

3.2 Architectural Principles

3.2.1 Modularity

Each component, screen, and service is designed as a self-contained module with clear responsibilities. This approach enables:

- Easy maintenance and debugging
- Code reusability across different parts of the application
- Independent testing of individual modules
- Simplified onboarding for new developers

3.2.2 Separation of Concerns

The architecture strictly separates:

- **Presentation Layer** (components/screens): UI rendering and user interactions
- **Business Logic Layer** (services/utils): Data processing and business rules
- **Data Layer** (data/mockData): Data structures and persistence

3.2.3 Component Hierarchy

- **Atomic Components:** Basic UI elements (buttons, inputs, cards)
- **Composite Components:** Combinations of atomic components
- **Screen Components:** Full-page views incorporating multiple components
- **Container Components:** Logic-heavy components managing state and data flow

3.2.4 Scalability Considerations

The structure is designed to accommodate future growth:

- Easy addition of new screens and features
- Minimal impact when modifying existing functionality
- Clear patterns for implementing new requirements
- Support for future backend integration

---

4. Development Procedures & Workflows

4.1 Initial Environment Setup

4.1.1 Prerequisites Verification

Step 1: Node.js Installation

- Download and install Node.js LTS version from nodejs.org
- Verify installation by running:
- `node --versionnpm --version`
- Expected output: Node v18.x.x or higher, npm v9.x.x or higher

Step 2: Development Tools Installation

- Install Visual Studio Code from code.visualstudio.com
- Install Android Studio from developer.android.com (if using emulator)
- Install Expo Go app on Android device from Google Play Store (if using physical device)

Step 3: Git Configuration

- Verify Git installation: `git --version`
- Configure user information:
- `git config --global user.name "Your Name"git config --global user.email "your.email@example.com"`

4.1.2 Project Setup

Step 1: Repository Clone

```
git clone [repository-url]
cd SplitFlowAndroid
```

Step 2: Dependency Installation

```
npm install
```

This command reads package.json and installs all required dependencies into the node_modules directory. The process typically takes 2-5 minutes depending on network speed.

Step 3: Environment Verification

```
npx expo --version
```

Confirms Expo CLI is properly installed and accessible.

4.2 Development Server Operations

4.2.1 Standard Development Server Launch

Command:

```
npx expo start
```

Process:

1. Metro bundler initializes and compiles JavaScript code
2. Development server starts on local network
3. QR code is generated in terminal
4. Server listens for connections from Expo Go or emulator

Console Output Indicators:

- "Metro waiting on..." - Server ready
- QR code display - Ready for device connection
- Build progress percentage - Compilation status

Interactive Commands:

- Press `a` - Open on Android emulator
- Press `i` - Open on iOS simulator (macOS only)
- Press `r` - Reload app
- Press `m` - Toggle menu

4.2.2 Tunnel Mode (Network-Independent Testing)

Command:

```
npm run tunnel
```

Description: This command executes `expo start --tunnel`, which utilizes Ngrok to create a secure public URL for the local development server. This is particularly useful when:

- Developer and testing devices are on different Wi-Fi networks
- Corporate firewalls block local network access
- Remote testing is required
- Local network has strict security policies

Process:

1. Ngrok client establishes secure tunnel
2. Public HTTPS URL is generated
3. Traffic is forwarded to local development server
4. QR code contains public URL instead of local IP

Performance Considerations:

- Initial connection may take 10-15 seconds
- Slight latency increase compared to local connection
- Dependent on internet connection stability

4.3 Testing Protocols & Quality Assurance

4.3.1 Physical Device Testing (Primary Method)

Setup Process:

Step 1: Device Preparation

- Ensure Android device is connected to internet
- Download and install Expo Go from Google Play Store
- Enable "Stay awake" in developer options (optional, prevents screen timeout)

Step 2: Connection Establishment

1. Launch development server (`npx expo start` or `npm run tunnel`)
2. Open Expo Go app on Android device
3. Scan QR code displayed in terminal using Expo Go
4. Wait for bundle to download and compile (first load: 30-60 seconds)

Step 3: Testing Execution

- Navigate through all application screens
- Test user interactions (buttons, forms, gestures)
- Verify data display and calculations
- Test edge cases and error scenarios
- Monitor console for errors and warnings

Hot Reload Feature:

- Code changes automatically reflect on device
- Saves changes and wait 2-3 seconds for reload
- No need to rescan QR code for subsequent changes

4.3.2 Android Emulator Testing

Setup Process:

Step 1: Emulator Configuration in Android Studio

- Open Android Studio
- Navigate to Tools > AVD Manager
- Create new virtual device or use existing
- Recommended configuration: Pixel 5 with Android 11 (API 30) or higher

Step 2: Emulator Launch and Testing

1. Ensure emulator is running (can be started from Android Studio or command line)
2. Launch development server: `npx expo start`
3. Press `a` in terminal to automatically open app in emulator
4. Application compiles and launches in emulator window

Advantages of Emulator Testing:

- Consistent environment across development team

- Easy to test different screen sizes and Android versions
- Network condition simulation capabilities
- Faster iteration during UI development

Limitations:

- Performance may differ from physical devices
- Hardware features (camera, GPS) require additional configuration
- Higher system resource consumption

4.3.3 Testing Checklist

Functional Testing:

- [ ] User authentication (login/logout)
- [ ] Project creation and management
- [ ] Expense addition and approval workflow
- [ ] Member role management
- [ ] Data calculations and statistics
- [ ] Navigation between screens

UI/UX Testing:

- [ ] Responsive layout on different screen sizes
- [ ] Button and touch target accessibility
- [ ] Loading states and animations
- [ ] Error message display
- [ ] Visual consistency with design specifications

Performance Testing:

- [ ] Application launch time
- [ ] Screen transition smoothness
- [ ] Scroll performance with large data sets
- [ ] Memory usage monitoring

---

5. Core Functionality & Business Logic

5.1 Project Management System

5.1.1 Dashboard Overview

Purpose: The Dashboard serves as the central hub for users to access and monitor their investment projects.

Key Components:

Active Projects Display:

- Card-based layout showing all projects user is involved in
- Real-time status indicators (Active, Pending, Completed)
- Quick stats: Total investment, number of members, pending approvals
- Tap-to-navigate functionality for detailed project view

Investment Statistics Aggregation:

- Total portfolio value across all projects
- Individual vs collective contribution breakdown
- Monthly investment trends
- Return on investment calculations (future enhancement)

Recent Activity Timeline:

- Chronological feed of project updates
- Expense approvals and rejections
- New member additions
- Role changes and administrative actions
- Timestamp display (relative time: "2 hours ago")

Quick Access Navigation:

- Fast action buttons: Add Expense, Create Project, View Notifications
- Search functionality for finding specific projects
- Filter options: Active/Archived, By Date, By Investment Amount

5.1.2 Project Hierarchical Structure

Administrative Roles:

- **Project Admin:** Full control over project settings, member management, and critical decisions
- **Permissions:** Add/remove members, change member roles, modify project details, final approval authority
- **Restrictions:** Cannot be removed from project while serving as admin
- **Transfer:** Admin role can be transferred to another active member

Investor Classifications:

Active Investors:

- Full participation rights in project decisions
- Voting privileges on expense approvals
- Can submit expenses
- Access to all project financial data
- Can view and download reports

Passive Investors:

- View-only access to project information
- Cannot vote on expenses or decisions
- Can contribute funds but cannot approve spending
- Suitable for silent partners or limited involvement members
- Can be promoted to Active status by Admin

Ledger Organization:

Main Ledger:

- Primary financial record for the project
- Tracks all income and expenses
- Maintains running balance
- Generates financial summaries

Sub-Ledgers:

- Category-specific financial tracking (Marketing, Operations, Development, etc.)
- Enables detailed expense categorization
- Supports budget allocation and monitoring
- Roll-up into main ledger for comprehensive view

5.2 Expense Management & Tracking

5.2.1 Add Spending Workflow

User Interface Design:

The Add Spending screen provides a comprehensive form for recording expenses with the following input fields:

Amount Field:

- Numeric keyboard input
- Currency formatting (automatic comma insertion)
- Validation: Must be greater than zero
- Maximum limit: Configurable per project
- Decimal precision: Two decimal places

Description Field:

- Text input for expense details
- Character limit: 200 characters
- Placeholder: "Enter expense description..."
- Required field validation
- Multi-line support for detailed explanations

Category Selection:

- Dropdown menu with two primary options:
    - **Service:** Intangible expenses (consulting, subscriptions, licenses)
    - **Product:** Tangible purchases (equipment, materials, inventory)
- Future enhancement: Custom category creation
- Default selection: None (forces user selection)

Investment Type Selection:

This critical field determines who is funding the expense:

Option 1: Self Account

- Indicates current user is personally funding the expense
- Automatically attributes expense to user's contribution record
- Increases user's total contribution to project
- Updates user's standing on contribution leaderboard

Option 2: Funded By [Member Name]

- Dynamic dropdown populated with all project members
- Allows recording of expenses paid by other members
- Common use cases:
    - Admin funding expenses on behalf of project
    - One member making purchase for group reimbursement
    - Recording historical expenses
- Maintains accurate attribution of contributions

Receipt Attachment (Future Enhancement):

- Image upload from camera or gallery
- PDF document attachment
- Multiple file support
- File size limit: 5MB per attachment

5.2.2 Approval System Architecture

Submission Process:

Step 1: Expense Creation

- User completes Add Spending form
- Validation checks are performed
- Expense is created with "Pending" status
- Timestamp and submitter information recorded

Step 2: Notification Distribution

- Push notifications sent to all Active members
- Email notifications (if enabled)
- In-app notification badge update

- Notification content: Amount, Description, Submitter

Step 3: Pending Queue Management

- Expense appears in "Pending Approvals" section
- Visible to all project members
- Different views for Active (actionable) vs Passive (informational)
- Sorting: Most recent first

Voting Mechanism:

Active Member Voting:

- Each Active member receives voting buttons: Approve / Reject
- Vote is recorded with timestamp
- Real-time progress indicator shows approval status
- Example: "3 of 5 members approved"

Approval Threshold:

- Requires unanimous approval from all Active members
- Alternative configuration: Simple majority (future enhancement)
- Admin has veto power (configurable)

Vote Outcomes:

Approved:

- Expense moves from "Pending" to "Approved" status
- Amount is deducted from project balance
- Contribution attributed to funding member
- Ledger entry created
- Confirmation notification sent to all members

Rejected:

- Expense marked as "Rejected"
- Reason for rejection captured (optional comment)
- Notification sent to submitter
- Expense remains in history for audit purposes
- Can be resubmitted with modifications

Passive Member Restrictions:

Passive members have limited interaction with expenses:

- **Can View:** Expense details, amounts, descriptions, voting progress
- **Cannot:** Approve, reject, or comment on pending expenses
- **Rationale:** Ensures decision-making remains with actively involved members
- **Notification:** Passive members receive informational notifications only

Timeout Handling:

- Configurable approval deadline (default: 72 hours)
- Automated reminders sent at 24 and 48 hours
- Expired approvals require Admin intervention
- Options: Auto-approve, auto-reject, or manual review

5.3 Member Administration & Role Management

5.3.1 Member Role Control

Admin Capabilities:

Role Toggle Functionality:

- Admin can change any member between Active and Passive status
- Toggle switch UI in Member Management screen
- Immediate effect upon confirmation
- Cannot toggle own status (prevents self-lockout)

Access Implications:

Active Status:

- Full voting rights on expense approvals
- Can submit expense requests
- Access to project management features
- Receives actionable notifications
- Counted in approval quorum

Passive Status:

- Read-only access to project data
- Cannot vote on expenses
- Can view but not create expenses
- Receives informational notifications only
- Excluded from approval quorum calculations

Role Change Notifications:

- Affected member receives notification of status change
- All project members notified of role modifications
- Audit trail maintained for compliance

Use Cases for Role Changes:

- Temporary absence: Active → Passive during member leave
- Reduced involvement: Stepping back from active participation

- Increased engagement: Passive → Active when ready to participate
- Disciplinary action: Removing voting rights while maintaining membership

5.3.2 Contribution Tracking & Analytics

Calculation Methodology:

Self-Funded Contributions:

- Sum of all expenses where member selected "Self Account"
- Includes both approved and pending expenses (with status indicators)
- Real-time updates as new expenses are added

Admin-Funded Contributions:

- Sum of all expenses where Admin was selected as funder
- Separately tracked for transparency
- Helps identify subsidized vs independent members

Funding Source Determination:

- Algorithm analyzes all member expenses
- Calculates percentage of self-funded vs admin-funded
- Threshold: >50% self-funded = "Self Funded" badge
- ≤50% self-funded = "Funded by [Admin Name]" badge

Leaderboard Display:

Ranking Criteria:

- Primary: Total contribution amount (descending order)
- Secondary: Number of approved expenses
- Tertiary: Join date (earlier members ranked higher in ties)

Visual Indicators:

- Position number (1st, 2nd, 3rd with special icons)
- Member name and profile picture
- Total contribution with currency formatting
- Funding status badge
- Progress bar showing contribution relative to top contributor

Leaderboard Features:

- Filterable by time period (All time, This year, This month)
- Exportable as PDF report
- Shareable within project team
- Privacy settings (Admin can hide from Passive members)

Analytics Dashboard:

- Contribution trends over time (line graph)
- Category-wise spending breakdown (pie chart)
- Member comparison matrix
- Projected contributions based on historical data

5.4 Data Management & State Handling

5.4.1 Mock Data Implementation

Purpose and Rationale:

During the development phase, SplitFlow utilizes a mock data structure to simulate backend responses. This approach provides several advantages:

Development Benefits:

- Enables frontend development without backend dependencies
- Allows comprehensive UI testing with realistic data scenarios
- Facilitates rapid prototyping and iteration
- Supports offline development environments

File Location: `src/data/mockData.js`

Data Structure Components:

User Objects:

```
{
  id: "user_001",
  name: "John Doe",
  email: "john.doe@example.com",
  role: "admin",
  profilePicture: "url_to_image",
  joinDate: "2025-01-15"
}
```

Project Objects:

```
{
  id: "project_001",
  name: "Mobile App Development",
  totalInvestment: 250000,
  members: [array of member IDs],
  status: "active",
  createdDate: "2025-01-01",
  admin: "user_001"
}
```

Expense Objects:

```
{
```

```
  id: "expense_001",
  projectId: "project_001",
  amount: 5000,
  description: "UI/UX Design Services",
  category: "service",
  fundedBy: "user_001",
  submittedBy: "user_002",
  status: "pending",
  submittedDate: "2025-02-05",
  approvals: [array of user IDs who approved],
  rejections: [array of user IDs who rejected]
}
```

Data Relationships:

- Normalized structure with ID references
- Maintains referential integrity
- Supports efficient querying and filtering
- Mimics actual database schema

Mock Data Updates:

- Regular synchronization with backend schema
- Version control for data structure changes
- Documentation of data relationships
- Sample data covers edge cases and scenarios

5.4.2 State Management Strategy

React Hooks Implementation:

useState Hook:

- Manages local component state
- Tracks form inputs, UI toggles, and temporary data
- Re-renders component when state changes
- Example: Managing form field values in Add Spending screen

```
const [amount, setAmount] = useState('');
const [description, setDescription] = useState('');
const [category, setCategory] = useState(null);
```

useEffect Hook:

- Handles side effects and data fetching
- Performs calculations when dependencies change
- Cleanup operations when components unmount
- Example: Recalculating total investment when expenses update

```
useEffect(() => {
  const total = expenses
    .filter(e => e.status === 'approved')
    .reduce((sum, e) => sum + e.amount, 0);
```

```
  setTotalInvestment(total);
}, [expenses]);
```

State Processing Patterns:

Derived State:

- Calculations performed from primary state
- Leaderboard rankings computed from contribution data
- Approval percentages calculated from voting status
- Avoids state duplication and ensures consistency

Optimistic Updates:

- UI updates immediately upon user action
- Provides responsive user experience
- Reverts if backend operation fails
- Example: Instantly showing expense in pending list

State Synchronization:

- Periodic polling for latest data (development phase)
- Websocket integration for real-time updates (future)
- Conflict resolution strategies
- Data consistency checks

Performance Optimization:

- Memoization of expensive calculations (useMemo)
- Callback optimization (useCallback)
- Lazy loading of components
- Virtual scrolling for large lists

Global State Management (Future Enhancement):

- Context API for app-wide state
- Redux/MobX consideration for complex state
- Persistent state with AsyncStorage
- State hydration on app restart

---

6. Version Control & Deployment

6.1 Git Workflow & Best Practices

6.1.1 Standard Commit Process

Step 1: Review Changes Before staging changes, review all modified files:

```
git status
```

This displays all changed, added, and deleted files. Verify that only intended changes are present.

Step 2: Stage Changes

```
git add .
```

Stages all changes in the current directory and subdirectories.

Alternative - Selective Staging:

```
git add src/screens/investor/DashboardScreen.js
git add src/components/Theme.js
```

Stages specific files only, useful when committing logically related changes separately.

Step 3: Commit with Descriptive Message

```
git commit -m "feat: add expense approval voting mechanism"
```

Commit Message Convention (Conventional Commits):

Format: `<type>: <description>`

Types:

- **feat:** New feature addition
  - Example: `feat: implement member leaderboard`
- **fix:** Bug fix
  - Example: `fix: correct expense calculation in dashboard`
- **docs:** Documentation changes
  - Example: `docs: update README with setup instructions`
- **style:** Code style changes (formatting, no logic change)
  - Example: `style: format DashboardScreen with Prettier`
- **refactor:** Code restructuring without functionality change
  - Example: `refactor: extract expense logic to service layer`
- **test:** Adding or modifying tests
  - Example: `test: add unit tests for expense validation`
- **chore:** Maintenance tasks
  - Example: `chore: update dependencies to latest versions`

Description Guidelines:

- Use imperative mood ("add" not "added" or "adds")
- Keep under 72 characters
- Be specific and descriptive
- Reference issue numbers when applicable

Step 4: Push to Remote Repository

```
git push origin main
```

Pushes committed changes to the 'main' branch on the remote repository (GitHub).

Pre-Push Checklist:

- [ ] All changes committed
- [ ] Commit messages follow convention
- [ ] Local testing completed successfully
- [ ] No sensitive data (API keys, passwords) in commits
- [ ] Pull latest changes to avoid conflicts

6.1.2 Branch Management Strategy

Main Branch:

- Contains production-ready code
- Protected branch (requires pull request for changes)
- Deployable at any time
- Tagged with version numbers

Development Branch (Future Implementation):

- Integration branch for features
- Regular merges from feature branches
- Pre-release testing environment
- Merged to main upon stable release

Feature Branches (Future Implementation):

- Created for each new feature
- Naming: `feature/expense-approval-system`
- Merged via pull request after code review
- Deleted after successful merge

Hotfix Branches (Future Implementation):

- For critical production bug fixes
- Naming: `hotfix/fix-login-crash`
- Direct merge to main and development
- Immediate deployment after testing

6.1.3 Collaboration Protocols

Pull Request Process:

1. Create feature branch from latest main
2. Implement changes with regular commits
3. Push feature branch to remote
4. Open pull request with description

5. Request code review from team members
6. Address review comments
7. Merge upon approval

Code Review Guidelines:

- Check code quality and adherence to standards
- Verify functionality through testing
- Ensure no breaking changes
- Validate commit message conventions
- Approve or request changes

Conflict Resolution:

- Pull latest main branch regularly
- Resolve merge conflicts locally
- Test thoroughly after conflict resolution
- Communicate with team about major conflicts

6.2 Versioning Strategy & Release Management

6.2.1 Semantic Versioning

Current Version: 1.0.3
Version Code: 4

Format: MAJOR.MINOR.PATCH

Version Component Definitions:

MAJOR (1.x.x):

- Incompatible API changes
- Complete redesign or architecture change
- Breaking changes requiring user migration
- Example: 1.0.3 → 2.0.0 (new authentication system)

MINOR (x.0.x):

- New features added in backward-compatible manner
- Functionality enhancements
- Non-breaking improvements
- Example: 1.0.3 → 1.1.0 (new reporting feature)

PATCH (x.x.3):

- Backward-compatible bug fixes
- Performance improvements

- Minor UI refinements
- Example: 1.0.3 → 1.0.4 (fix expense calculation bug)

Version Code:

- Integer incremented with each release
- Used for Android app updates
- Independent of semantic version
- Current: 4 (increments: 4 → 5 → 6...)

6.2.2 Version Update Protocol

Files Requiring Version Updates:

app.json:

```
{
  "expo": {
    "name": "SplitFlow",
    "version": "1.0.3",
    "android": {
      "versionCode": 4
    }
  }
}
```

build.gradle (Android specific):

```
android {
    defaultConfig {
        versionCode 4
        versionName "1.0.3"
    }
}
```

package.json:

```
{
  "name": "splitflow-android",
  "version": "1.0.3"
}
```

Update Checklist:

- [ ] Determine version type (major/minor/patch)
- [ ] Update version in app.json
- [ ] Update versionCode in app.json
- [ ] Update version in package.json
- [ ] Update build.gradle if applicable
- [ ] Document changes in CHANGELOG.md
- [ ] Commit with message: `chore: bump version to 1.0.4`
- [ ] Create Git tag: `git tag v1.0.4`
- [ ] Push tag: `git push origin v1.0.4`

6.2.3 Release Process

Pre-Release Preparation:

1. Complete all planned features and fixes
2. Run comprehensive testing suite
3. Update documentation
4. Prepare release notes
5. Finalize version number

Build Generation:

Development Build (Testing):

```
eas build --profile preview --platform android
```

Production Build (Release):

```
eas build --profile production --platform android
```

Release Checklist:

- [ ] All tests passing
- [ ] No critical bugs
- [ ] Documentation updated
- [ ] Version numbers updated
- [ ] Release notes prepared
- [ ] Stakeholder approval obtained
- [ ] Backup of previous version created

Deployment Steps:

1. Generate production APK
2. Test APK on multiple devices
3. Upload to internal testing track
4. Conduct UAT (User Acceptance Testing)
5. Promote to production track
6. Monitor for post-release issues

Rollback Plan:

- Keep previous version APK accessible
- Document rollback procedure
- Maintain version history
- Quick response protocol for critical issues

---

7. Troubleshooting & Maintenance

7.1 Common Issues & Resolution Procedures

7.1.1 Network Connectivity Issues

Issue: "Network request failed" Error

Symptoms:

- Unable to load app on Expo Go
- QR code scan succeeds but app doesn't load
- Intermittent connection drops during development

Root Causes:

- Developer machine and testing device on different networks
- Firewall blocking Metro bundler port
- VPN interference
- Network instability

Resolution Steps:

Solution 1: Enable Tunnel Mode

```
npm run tunnel
```

Explanation: Tunnel mode uses Ngrok to create a public URL, bypassing local network restrictions. This is the most reliable solution for network issues.

Process:

1. Stop current development server (Ctrl+C)
2. Run `npm run tunnel`
3. Wait for Ngrok tunnel to establish (10-15 seconds)
4. Scan new QR code with Expo Go
5. Application should load successfully

Solution 2: Verify Network Configuration

Check Network Connectivity:

```
ipconfig  # Windows
ifconfig  # Mac/Linux
```

- Ensure both devices show same network subnet
- Example: Developer machine 192.168.1.100, device 192.168.1.105 ✓
- Different subnets: 192.168.1.100 vs 192.168.2.100 ✗

Check Firewall Settings:

- Windows: Allow Metro bundler through firewall (port 8081)

- Mac: System Preferences → Security & Privacy → Firewall → Allow Metro
- Temporarily disable firewall for testing (re-enable after)

Solution 3: Use Direct Connection (LAN)

```
npx expo start --lan
```

Forces Expo to use LAN IP address instead of localhost.

Verification:

- Successful connection shows "Building JavaScript bundle" in terminal
- Progress percentage indicates active loading
- App launches within 30-60 seconds on first load

7.1.2 Cache-Related Issues

Issue: Stale Code, Unexpected Behavior, Build Errors

Symptoms:

- Code changes not reflecting in app
- Persistent errors after bug fixes
- "Unable to resolve module" errors
- Inconsistent behavior between builds

Root Causes:

- Metro bundler cache corruption
- Node modules cache issues
- Outdated JavaScript bundles
- Watchman cache problems (Mac/Linux)

Resolution Steps:

Solution 1: Clear Metro Bundler Cache

```
npx expo start --clear
```

Explanation: Deletes Metro bundler cache and rebuilds JavaScript bundle from scratch.

When to Use:

- After modifying dependencies
- When code changes aren't appearing
- After git branch switches
- As first troubleshooting step

Solution 2: Complete Cache Purge

```
# Clear Expo cache
npx expo start -c

# Clear npm cache
npm cache clean --force

# Clear watchman cache (Mac/Linux)
watchman watch-del-all

# Restart development server
npx expo start --clear
```

Solution 3: Nuclear Option (Complete Reset)

```
# Delete node_modules
rm -rf node_modules

# Delete package-lock.json
rm package-lock.json

# Reinstall dependencies
npm install

# Start with clear cache
npx expo start --clear
```

When to Use Nuclear Option:

- Persistent unexplainable errors
- After major dependency updates
- When switching between branches with different dependencies
- Before reporting bugs to ensure clean environment

Prevention:

- Regular cache clearing during active development
- Clean builds before major releases
- Document when cache clearing resolves issues

7.1.3 Build Failures

Issue: APK Build Fails or Crashes

Symptoms:

- Build process terminates with errors
- Generated APK crashes on launch
- "Build failed" messages in EAS
- Dependency resolution errors

Common Error Types and Solutions:

Error Type 1: Dependency Conflicts

Symptoms:

```
npm ERR! peer dependency conflict
npm ERR! Could not resolve dependency
```

Solution:

```
# Check for dependency conflicts
npm ls

# Update conflicting packages
npm update [package-name]

# Use legacy peer deps (temporary fix)
npm install --legacy-peer-deps
```

Error Type 2: Out of Memory

Symptoms:

```
FATAL ERROR: Ineffective mark-compacts near heap limit
JavaScript heap out of memory
```

Solution:

```
# Increase Node.js memory limit
export NODE_OPTIONS="--max-old-space-size=4096"

# Then rebuild
npx expo start --clear
```

Error Type 3: Build Configuration Issues

Symptoms:

- Build succeeds but APK won't install
- Version code conflicts
- Signing errors

Solution:

1. Verify `app.json` configuration
2. Check `build.gradle` for syntax errors
3. Ensure version codes are incrementing
4. Validate signing credentials

Complete Troubleshooting Workflow:

1. Read error message carefully (80% of solutions are in the error text)
2. Clear cache and retry
3. Check recent code changes
4. Verify dependency versions
5. Search error message in Expo forums

6. Check GitHub issues for known bugs
7. Ask for help with detailed error logs

7.1.4 Performance Issues

Issue: Slow App Performance, Lag, Crashes

Symptoms:

- Sluggish screen transitions
- Delayed response to user inputs
- Frame drops during animations
- Memory warnings
- App crashes after extended use

Diagnostic Tools:

React Native Performance Monitor:

- Enable in Expo Go: Shake device → "Show Performance Monitor"
- Metrics: JS frame rate, UI frame rate, memory usage
- Target: 60 FPS for smooth experience

Console Logging:

```
console.time('expenseCalculation');
// ... expensive operation
console.timeEnd('expenseCalculation');
```

Resolution Strategies:

Optimize Re-renders:

- Use `React.memo()` for expensive components
- Implement `useMemo()` for costly calculations
- Use `useCallback()` for function props
- Avoid inline function definitions in render

Optimize Lists:

- Implement `FlatList` instead of `ScrollView` for large datasets
- Use `getItemLayout` for fixed-height items
- Enable `removeClippedSubviews` on Android
- Implement pagination for very large lists

Image Optimization:

- Compress images before including in assets
- Use appropriate image dimensions
- Implement lazy loading for off-screen images

- Consider CDN for remote images

Memory Management:

- Clean up event listeners in `useEffect` cleanup
- Avoid memory leaks from uncancelled promises
- Clear intervals and timeouts
- Remove unused dependencies

7.2 Feature Implementation Guidelines

7.2.1 Structured Development Process

Step 1: Requirements Analysis

Activities:

- Review feature requirements document
- Identify data structures needed
- Determine UI components required
- Assess impact on existing features
- Estimate development time

Deliverables:

- Feature specification document
- Data model design
- UI mockups or wireframes
- Technical approach document

Step 2: Data Structure Definition

Location: `src/data/mockData.js`

Process:

1. Identify new data entities required
2. Define object schemas with all properties
3. Establish relationships with existing data
4. Create sample data for testing
5. Document data structure with comments

Example - Adding "Comments" Feature:

```
// Add to mockData.js
export const comments = [
  {
    id: "comment_001",
    expenseId: "expense_001",  // Reference to expense
```

```
    userId: "user_001",          // Comment author
    text: "This expense seems high, can you provide more details?",
    timestamp: "2025-02-08T14:30:00Z",
    isResolved: false
  }
];
```

Data Integrity Checks:

- Ensure referential integrity (IDs match existing entities)
- Validate data types
- Test with edge cases (empty arrays, null values)
- Verify data consistency across related objects

Step 3: Component Development

Location: `src/components/`

Component Creation Process:

3a. Create Component File

```
# Create new component file
touch src/components/CommentCard.js
```

3b. Implement Component Logic

```
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

const CommentCard = ({ comment, user }) => {
  return (
    <View style={styles.container}>
      <Text style={styles.author}>{user.name}</Text>
      <Text style={styles.text}>{comment.text}</Text>
      <Text style={styles.timestamp}>
        {new Date(comment.timestamp).toLocaleString()}
      </Text>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    padding: 12,
    marginVertical: 6,
    backgroundColor: '#f5f5f5',
    borderRadius: 8,
  },
  author: {
    fontSize: 14,
    fontWeight: 'bold',
    marginBottom: 4,
  },
  text: {
    fontSize: 14,
    color: '#333',
```

```
    marginBottom: 8,
  },
  timestamp: {
    fontSize: 12,
    color: '#666',
  },
});

export default CommentCard;
```

Component Best Practices:

- Follow existing naming conventions
- Use functional components with hooks
- Extract reusable logic to custom hooks
- Implement PropTypes or TypeScript for type safety
- Add comments for complex logic

Step 4: Screen Implementation

Location: `src/screens/investor/` (or appropriate subdirectory)

Screen Creation Process:

4a. Create Screen File

```
touch src/screens/investor/ExpenseCommentsScreen.js
```

4b. Implement Screen Logic

```
import React, { useState, useEffect } from 'react';
import { View, FlatList, TextInput, Button } from 'react-native';
import CommentCard from '../../components/CommentCard';
import { comments as mockComments } from '../../data/mockData';

const ExpenseCommentsScreen = ({ route }) => {
  const { expenseId } = route.params;
  const [comments, setComments] = useState([]);
  const [newComment, setNewComment] = useState('');

  useEffect(() => {
    // Load comments for this expense
    const expenseComments = mockComments.filter(
      c => c.expenseId === expenseId
    );
    setComments(expenseComments);
  }, [expenseId]);

  const handleAddComment = () => {
    // Logic to add new comment
    const comment = {
      id: `comment_${Date.now()}`,
      expenseId,
      userId: 'current_user_id',
      text: newComment,
      timestamp: new Date().toISOString(),
      isResolved: false
```

```
    };
    setComments([...comments, comment]);
    setNewComment('');
  };

  return (
    <View style={{ flex: 1 }}>
      <FlatList
        data={comments}
        keyExtractor={item => item.id}
        renderItem={({ item }) => <CommentCard comment={item} />}
      />
      <TextInput
        value={newComment}
        onChangeText={setNewComment}
        placeholder="Add a comment..."
      />
      <Button title="Post Comment" onPress={handleAddComment} />
    </View>
  );
};

export default ExpenseCommentsScreen;
```

Screen Development Guidelines:

- Implement loading states for data fetching
- Add error handling and user feedback
- Ensure responsive design for different screen sizes
- Follow navigation patterns established in existing screens
- Implement proper keyboard handling for forms

Step 5: Navigation Registration

Location: `App.js`

Process:

5a. Import New Screen

```
import ExpenseCommentsScreen from
'./src/screens/investor/ExpenseCommentsScreen';
```

5b. Add Route to Stack Navigator

```
<Stack.Navigator>
  {/* Existing screens */}
  <Stack.Screen
    name="ExpenseComments"
    component={ExpenseCommentsScreen}
    options={{ title: 'Comments' }}
  />
</Stack.Navigator>
```

5c. Navigate from Existing Screen

```
// In ExpenseDetailScreen.js
import { useNavigation } from '@react-navigation/native';

const navigation = useNavigation();

// In button onPress
navigation.navigate('ExpenseComments', { expenseId: expense.id });
```

Navigation Best Practices:

- Use descriptive route names
- Pass minimal required parameters
- Configure headers appropriately
- Test deep linking (if applicable)
- Ensure proper back navigation behavior

Step 6: Testing & Validation

Testing Checklist:

- [ ] **Functionality Testing**
  - Feature works as specified
  - All user interactions respond correctly
  - Edge cases handled properly
- [ ] **UI/UX Testing**
  - Layout displays correctly on different screen sizes
  - Text is readable and properly styled
  - Colors match design specifications
  - Loading states appear appropriately
- [ ] **Integration Testing**
  - New feature integrates with existing functionality
  - No regressions in other features
  - Data flows correctly between screens
- [ ] **Performance Testing**
  - No noticeable lag or performance degradation
  - Smooth animations and transitions
  - Efficient re-rendering
- [ ] **Code Quality**
  - Code follows project conventions
  - Proper comments and documentation
  - No console warnings or errors
  - Linting passes without errors

Step 7: Documentation & Deployment

Documentation Updates:

- Update this SOP with new feature details
- Add code comments explaining complex logic
- Update README if new dependencies added
- Create feature documentation for end users

Deployment Process:

1. Commit changes with descriptive message
2. Create pull request (if using feature branches)
3. Code review by team member
4. Merge to main branch
5. Update version number if significant feature
6. Deploy to testing environment
7. Conduct user acceptance testing
8. Deploy to production

---

8. Quality Assurance & Standards

8.1 Code Quality Standards

8.1.1 Coding Conventions

JavaScript Style Guide:

- Use ES6+ features (arrow functions, destructuring, template literals)
- Prefer `const` over `let`, avoid `var`
- Use meaningful variable and function names
- Follow camelCase for variables and functions
- Follow PascalCase for components and classes

Component Structure:

```
// 1. Imports
import React, { useState, useEffect } from 'react';
import { View, Text } from 'react-native';

// 2. Component Definition
const MyComponent = ({ prop1, prop2 }) => {
  // 3. State declarations
  const [state, setState] = useState(initialValue);

  // 4. Effects
  useEffect(() => {
    // effect logic
  }, [dependencies]);

  // 5. Event handlers
  const handlePress = () => {
    // handler logic
  };

  // 6. Render
  return (
    <View>
      <Text>{prop1}</Text>
    </View>
  );
```

```
};

// 7. Styles
const styles = StyleSheet.create({
  // styles
});

// 8. Export
export default MyComponent;
```

File Organization:

- One component per file
- File name matches component name
- Group related files in subdirectories
- Keep file length reasonable (<300 lines)

8.1.2 Performance Optimization

React Optimization Techniques:

- Memoize expensive calculations with `useMemo`
- Optimize callbacks with `useCallback`
- Use `React.memo` for pure components
- Implement virtualized lists for large datasets
- Avoid unnecessary re-renders

Best Practices:

- Extract static data outside components
- Use keys properly in lists
- Avoid inline style objects
- Debounce rapid user inputs
- Lazy load images and components

8.1.3 Error Handling

Comprehensive Error Handling:

```
try {
  // Risky operation
  const result = await fetchData();
  setData(result);
} catch (error) {
  console.error('Error fetching data:', error);
  // User-friendly error message
  Alert.alert('Error', 'Failed to load data. Please try again.');
}
```

User Feedback:

- Display loading indicators during operations
- Show clear error messages

- Provide retry options
- Log errors for debugging
- Graceful degradation when features fail

8.2 Security Considerations

8.2.1 Data Protection

Sensitive Data Handling:

- Never commit API keys, passwords, or secrets to Git
- Use environment variables for configuration
- Implement proper authentication mechanisms
- Sanitize user inputs
- Validate all data before processing

Future Implementation:

- Secure storage for tokens (AsyncStorage encryption)
- HTTPS for all API communications
- JWT token management
- Biometric authentication
- Data encryption at rest

8.2.2 Code Security

Security Best Practices:

- Regular dependency audits: `npm audit`
- Update dependencies to patch vulnerabilities
- Validate and sanitize user inputs
- Implement proper authorization checks
- Follow OWASP mobile security guidelines

---

9. Maintenance Schedule & Procedures

9.1 Regular Maintenance Tasks

9.1.1 Weekly Activities

Code Review:

- Review all commits from the past week
- Ensure coding standards compliance
- Identify potential improvements

- Document technical debt

Dependency Check:

```
npm outdated
```

- Review outdated packages
- Plan updates for non-breaking changes
- Document breaking changes for major updates

Testing:

- Run full test suite
- Test on physical devices
- Verify critical user flows
- Check performance metrics

9.1.2 Monthly Activities

Security Audit:

```
npm audit
npm audit fix
```

- Address security vulnerabilities
- Update dependencies with security patches
- Review access controls
- Audit third-party libraries

Performance Review:

- Analyze app performance metrics
- Identify bottlenecks
- Optimize slow screens
- Review bundle size
- Monitor memory usage

Documentation Update:

- Update README with new features
- Revise this SOP with process improvements
- Update API documentation
- Review and update code comments

Backup Verification:

- Verify Git repository backups
- Test disaster recovery procedures
- Document current system state
- Archive old builds

9.1.3 Quarterly Activities

Code Quality Assessment:

- Comprehensive code review
- Refactoring opportunities
- Architecture evaluation
- Technical debt prioritization

Dependency Major Updates:

- Plan and execute major dependency updates
- Test thoroughly after updates
- Update documentation
- Train team on new features

Security Review:

- Comprehensive security audit
- Penetration testing (when applicable)
- Review access controls
- Update security documentation

Performance Optimization:

- Deep performance analysis
- Optimize critical paths
- Reduce bundle size
- Improve startup time

9.2 Documentation Maintenance

9.2.1 Documentation Requirements

Code Documentation:

- Inline comments for complex logic
- JSDoc style comments for functions
- README files in component directories
- API documentation

Process Documentation:

- This SOP updated with process changes
- Decision logs for architectural choices
- Troubleshooting guides
- Onboarding documentation for new developers

9.2.2 Change Documentation

Change Log Maintenance:

- Document all significant changes
- Include version numbers
- Categorize changes (features, fixes, breaking changes)
- Link to relevant commits and pull requests

Format:

```
## [1.0.4] - 2025-02-15
### Added
- Comment system for expense approvals
- Email notifications for pending approvals

### Fixed
- Calculation error in contribution leaderboard
- Navigation bug on Android 12

### Changed
- Updated dashboard layout for better usability
- Improved expense form validation
```

---

10. Appendices

Appendix A: Command Reference

Essential Development Commands

Project Setup:

```
# Clone repository
git clone [repository-url]

# Install dependencies
npm install

# Verify installation
npx expo --version
```

Development Server:

```
# Standard start
npx expo start

# Start with tunnel (network-independent)
npm run tunnel

# Start with cache clear
npx expo start --clear

# Start in LAN mode
npx expo start --lan
```

Testing:

```
# Open on Android emulator
# (Press 'a' in running Expo terminal)

# Open on iOS simulator (macOS only)
# (Press 'i' in running Expo terminal)

# Reload app
# (Press 'r' in running Expo terminal)
```

Cache Management:

```
# Clear Expo cache
npx expo start -c

# Clear npm cache
npm cache clean --force

# Clear watchman cache (Mac/Linux)
watchman watch-del-all

# Complete reset
rm -rf node_modules
rm package-lock.json
npm install
```

Dependency Management:

```
# Install package
npm install [package-name]

# Install dev dependency
npm install --save-dev [package-name]

# Update packages
npm update

# Check outdated packages
npm outdated

# Security audit
npm audit
npm audit fix
```

Git Commands:

```
# Check status
git status

# Stage all changes
git add .

# Stage specific file
git add path/to/file

# Commit
git commit -m "type: description"
```

```
# Push to remote
git push origin main

# Pull latest changes
git pull origin main

# Create branch
git checkout -b feature/feature-name

# Switch branch
git checkout branch-name

# View commit history
git log --oneline

# Create tag
git tag v1.0.4
git push origin v1.0.4
```

Build Commands:

```
# Development build
eas build --profile preview --platform android

# Production build
eas build --profile production --platform android

# Check build status
eas build:list
```

Appendix B: Useful Resources

Official Documentation

React Native:

- Official Docs: https://reactnative.dev/docs/getting-started
- API Reference: https://reactnative.dev/docs/components-and-apis
- Community Resources: https://reactnative.dev/community/overview

Expo:

- Documentation: https://docs.expo.dev/
- API Reference: https://docs.expo.dev/versions/latest/
- Forums: https://forums.expo.dev/

React Navigation:

- Documentation: https://reactnavigation.org/docs/getting-started
- API Reference: https://reactnavigation.org/docs/navigation-prop

Git:

- Documentation: https://git-scm.com/doc
- Interactive Tutorial: https://learngitbranching.js.org/

Learning Resources

JavaScript/ES6+:

- MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript
- JavaScript.info: https://javascript.info/

React:

- Official Tutorial: https://react.dev/learn
- React Hooks Guide: https://react.dev/reference/react

Mobile Development:

- Android Developer Guide: https://developer.android.com/guide
- Material Design Guidelines: https://material.io/design

Community & Support

Stack Overflow:

- React Native tag: https://stackoverflow.com/questions/tagged/react-native
- Expo tag: https://stackoverflow.com/questions/tagged/expo

GitHub:

- React Native Issues: https://github.com/facebook/react-native/issues
- Expo Issues: https://github.com/expo/expo/issues

Discord/Slack Communities:

- Reactiflux Discord: https://www.reactiflux.com/
- Expo Discord: https://discord.gg/expo

Appendix C: Troubleshooting Quick Reference

Quick Diagnostic Table

Symptom	Likely Cause	Quick Fix
Network request failed	Different networks	Use tunnel mode: `npm run tunnel`
Code changes not showing	Cache issues	Clear cache: `npx expo start --clear`
Module not found	Missing dependency	Run: `npm install`
Build fails	Dependency conflict	Delete node_modules, reinstall
App crashes on launch	Code error	Check console logs, fix errors
Slow performance	Too many re-renders	Optimize with memo/useMemo
White screen	Navigation error	Check navigation configuration
Cannot connect to device	Firewall blocking	Allow Metro bundler in firewall
Version mismatch	Outdated dependencies	Run: `npm update`
Out of memory	Large bundle size	Increase Node memory limit

Appendix D: Glossary of Terms

APK: Android Package Kit - Installation file format for Android apps

Bundle: Compiled JavaScript code ready for execution

Component: Reusable piece of UI in React

Dependency: External library or package used in the project

Hook: React function that lets you use state and other React features

Metro: JavaScript bundler used by React Native

Mock Data: Simulated data used during development

Props: Properties passed to React components

Repository: Storage location for project files (Git)

State: Data that changes over time in a component

Tunnel: Secure connection through Ngrok for cross-network testing

---

Revision History

Version	Date	Author	Changes
1.0.0	2025-01-15	Development Team	Initial SOP creation
1.0.1	2025-01-28	Development Team	Added troubleshooting section
1.0.2	2025-02-03	Development Team	Updated technology stack details
1.0.3	2025-02-09	Development Team	Comprehensive elaboration and restructuring

---

End of Document