EvenUp - Software Design Document (SDGr05)

Group 05
Anshika Gupta - CS22BTECH11007
Yash Patel - CS22BTECH11047
Monish Asawa - MA22BTECH11014
Anirudh Saikrishnan - CS22BTECH11004

March 15, 2025

Contents

Chapter 1

Software Architecture

1.0.1 Component-and-Connector View

The figure below depicts our system's high-level component-and-connector (C&C) architecture. We separate responsibilities among the **Client**, the **Server** (further divided into **Business Logic & Auth** and **WebSocket Manager**), and the **Database**.

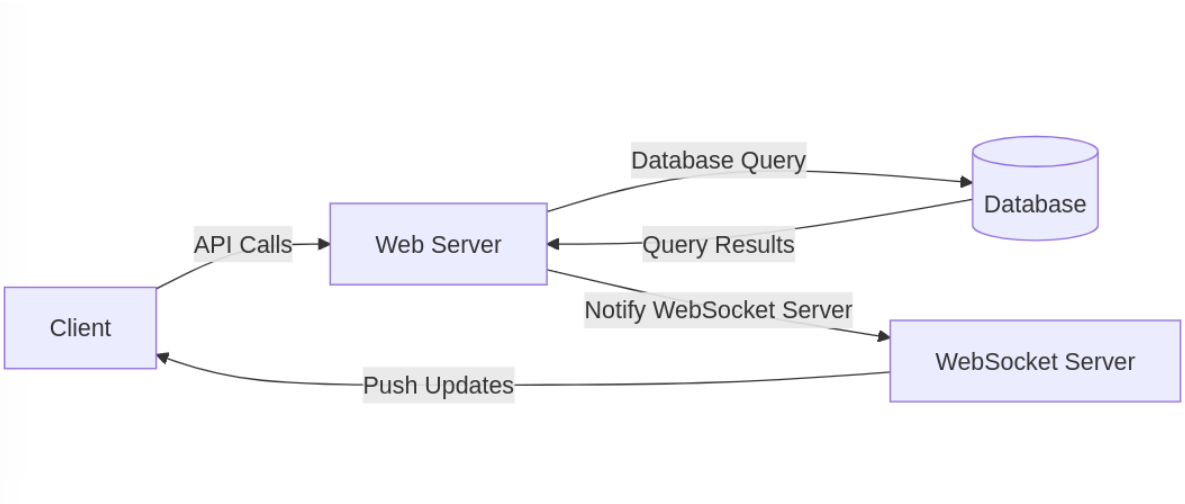


Figure 1.1: High-Level Component and Connector Architecture

1.0.2 Explanation of the Architecture

The system is composed of four primary components:

- **Client:** A mobile application responsible for sending requests to the server and rendering the user interface. It also maintains a WebSocket connection for real-time updates.
- **Business Logic & Auth:** A server-side component that processes incoming requests, handles authentication, and executes core application logic. It communicates with the database to store and retrieve persistent data.
- **WebSocket Manager:** Manages real-time connections (via WebSockets). It receives notifications from the Business Logic component when relevant data changes occur and pushes updates to connected clients.

- **Database:** Stores all persistent information, including user accounts, groups/private-split expenses, and transaction records. Queries are initiated by the Business Logic & Auth layer.

Connectors between these components include:

- **HTTP/REST** between Client and Business Logic.
- **SQL queries** between Business Logic and Database.
- **WebSocket** connections between Client and WebSocket Manager.

This design ensures clear separation of concerns: the Client focuses on presentation and user interaction, while the Server handles application logic, real-time updates, and data persistence.

1.1 ATAM Analysis

Below, we apply the Architecture Tradeoff Analysis Method (ATAM) to evaluate how this architecture meets various quality attributes under common usage scenarios. We focus on some scenarios that highlight potential tradeoffs and design decisions.

1.1.1 Scenarios and Evaluation

1. **High Concurrent Users (Performance):**
    - **Stimulus:** 1000+ concurrent users send HTTP requests and maintain WebSocket connections.
    - **Tradeoff:** Increased operational complexity vs. better performance and responsiveness under heavy load.
2. **Authentication Failure (Security):**
    - **Stimulus:** Malicious user tries to log in with invalid credentials or brute-force attempts.
    - **Response:** Business Logic & Auth module detects repeated failures and locks out accounts after N attempts. WebSocket Manager rejects unauthorized connections.
    - **Tradeoff:** Usability vs. stricter security policies.
3. **Secure Data Transmission (Security):**
    - **Stimulus:** An attacker attempts to intercept or sniff sensitive data (e.g., login credentials, personal info) in transit between the client and the server.
    - **Response:** All traffic is encrypted using TLS/HTTPS. The Go server enforces secure sessions and tokens. PostgreSQL connections also use SSL where possible.
    - **Tradeoff:** Slight overhead in encrypting/decrypting data vs. significantly improved security and protection against eavesdropping.

4. **Real-Time Updates (Responsiveness):**

   - **Stimulus:** Multiple users edit the same data; changes must appear instantly on all connected clients.
   - **Response:** WebSocket Manager broadcasts update events to subscribed clients; minimal round-trip time ensures near-instant synchronization.
   - **Tradeoff:** Additional server overhead for maintaining WebSocket connections vs. improved user experience with live updates.

5. **Large Data Requests (Performance / Scalability):**

   - **Stimulus:** Users request large datasets, potentially spanning millions of rows in PostgreSQL.
   - **Response:** The server processes queries in smaller segments and returns partial results. The Flutter client fetches each chunk sequentially and updates the UI incrementally as data arrives. This approach reduces memory usage on both client and server while improving perceived responsiveness.
   - **Tradeoff:** Additional complexity for chunked retrieval and incremental rendering versus faster response times and a more scalable solution for handling large datasets.

1.1.2 Summary of ATAM Findings

Overall, the architecture's separation into Business Logic, WebSocket Manager, and a dedicated Database ensures each concern is addressed in a focused manner. The Business Logic & Auth component centralizes security and data processing, while the WebSocket Manager handles real-time updates without overburdening the main request path. PostgreSQL provides robust transactional guarantees.

The analyzed scenarios highlight how the system deals with security (e.g., preventing brute-force attacks, securing data in transit), performance (e.g., chunked data retrieval), and responsiveness (e.g., near-instant updates via WebSockets). Each scenario illustrates a tradeoff between added complexity (e.g., implementing chunked retrieval or managing many WebSocket connections) and improved user experience or robustness.

Chapter 2

Data Flow Diagrams

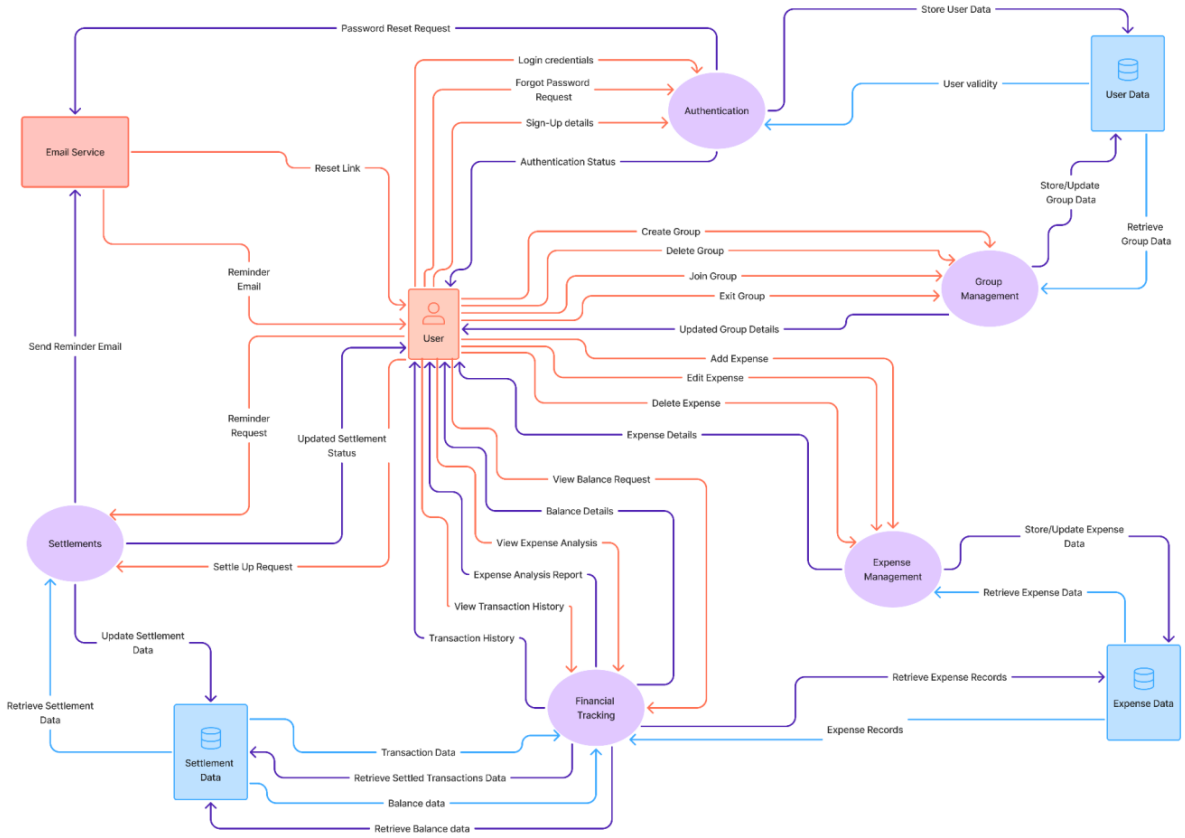


Figure 2.1: DFD

Table 2.1: Most Abstract Inputs (mai) and Most Abstract Outputs (mao)

Module	**Most Abstract Input (mai)**	**Most Abstract Output (mao)**
Sign Up	New user account details	Verified user account (created user)
Login	User credentials (username/password)	Authenticated session token or verified user account
Change Password	Verified user ID, old password, new password	Confirmation of updated credentials
Add Group	Verified user ID, group details (name, description)	New group record (group ID)
Add Expense	Verified user ID, group ID, expense details (amount, description, etc.)	Updated group ledger
Settle Expense	Verified user ID, group ID, settlement details	Updated balances for all group members
Fetch Groups	Verified user ID	List of groups (IDs, names)
Fetch Group Data	Verified user ID, group ID	Detailed expense records (who owes what, group totals, etc.)
Log Out	Verified session token (or verified user ID)	Session invalidation (user logged out)
Real-Time Update	Internal server event (e.g., data change in a group)	WebSocket push notification (updated balances/UI)

Chapter 3

Structure Charts

3.1 First-Level Factored Modules

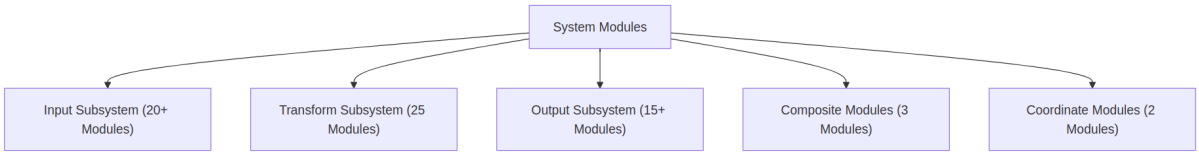


Figure 3.1: First-level Factored Modules

3.2 Factored Input Modules

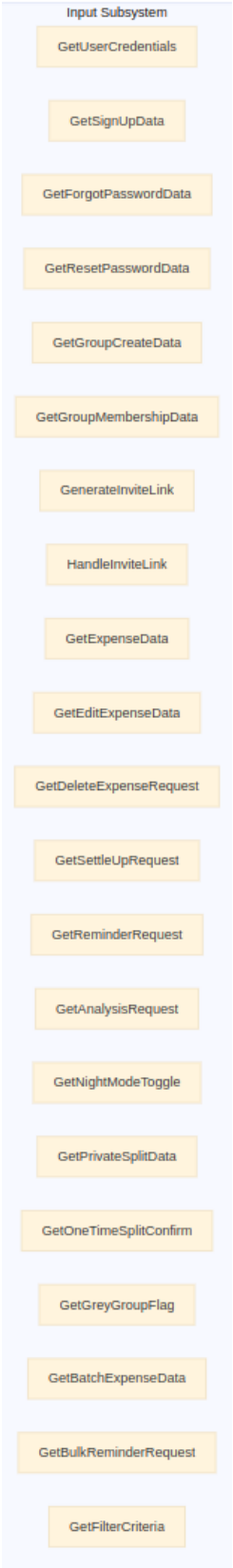
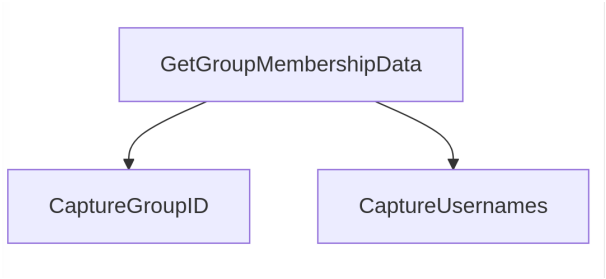
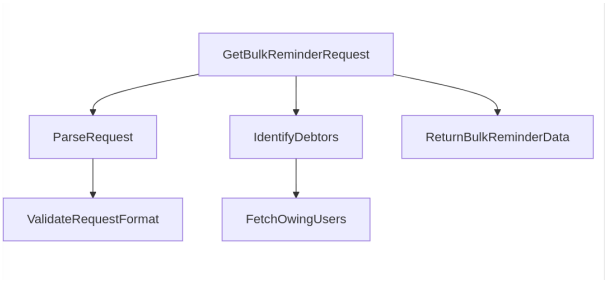
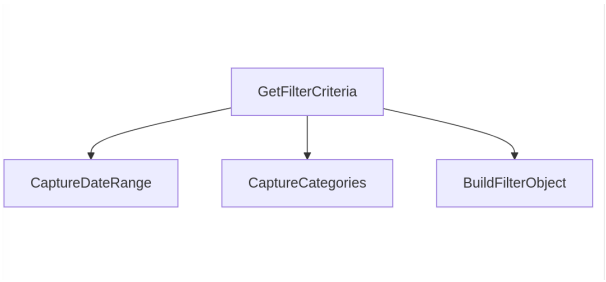


Figure 3.2: Factored input modules

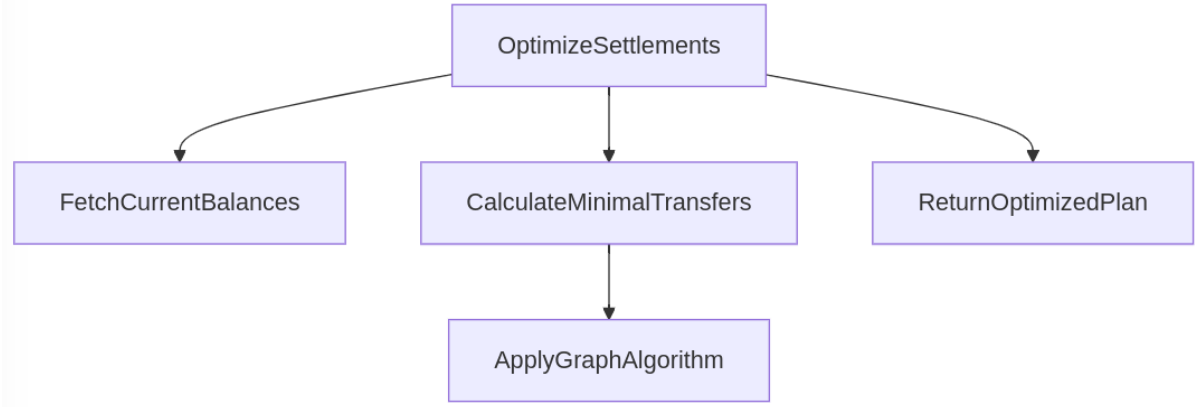

GetGroupMembershipData
CaptureGroupID
CaptureUsernames

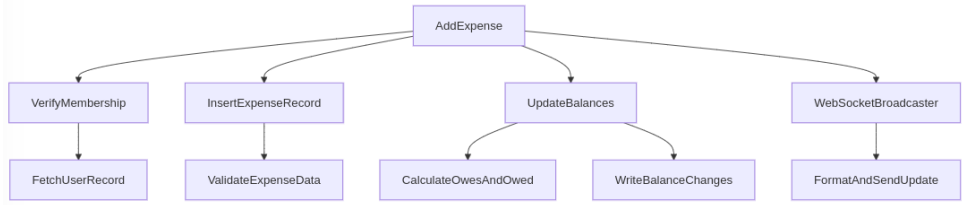

GetBulkReminderRequest
ParseRequest
IdentifyDebtors
ReturnBulkReminderData
ValidateRequestFormat
FetchOwingUsers

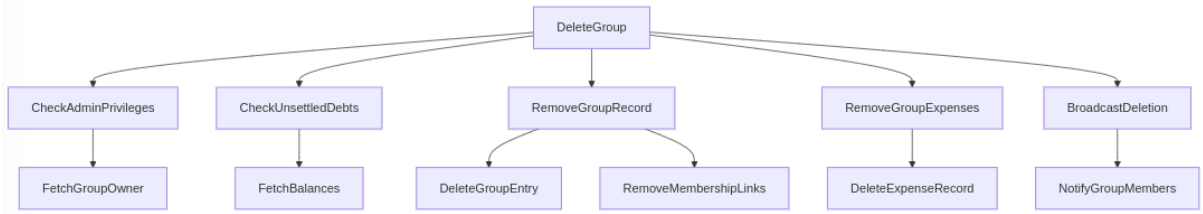

GetFilterCriteria
CaptureDateRange
CaptureCategories
BuildFilterObject

3.3 Factored Transform Modules

Figure 3.3: Factored Transform modules


OptimizeSettlements
FetchCurrentBalances
CalculateMinimalTransfers
ReturnOptimizedPlan
ApplyGraphAlgorithm


AddExpense
VerifyMembership
InsertExpenseRecord
UpdateBalances
WebSocketBroadcaster
FetchUserRecord
ValidateExpenseData
CalculateOwesAndOwed
WriteBalanceChanges
FormatAndSendUpdate


DeleteGroup
CheckAdminPrivileges
CheckUnsettledDebts
RemoveGroupRecord
RemoveGroupExpenses
BroadcastDeletion
FetchGroupOwner
FetchBalances
DeleteGroupEntry
RemoveMembershipLinks
DeleteExpenseRecord
NotifyGroupMembers

3.4 Factored Output Modules

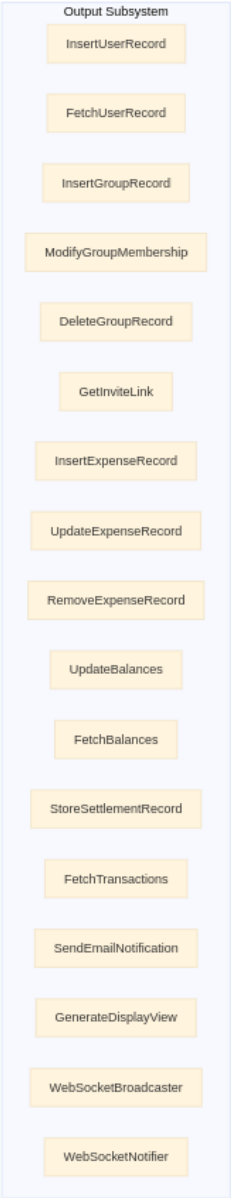
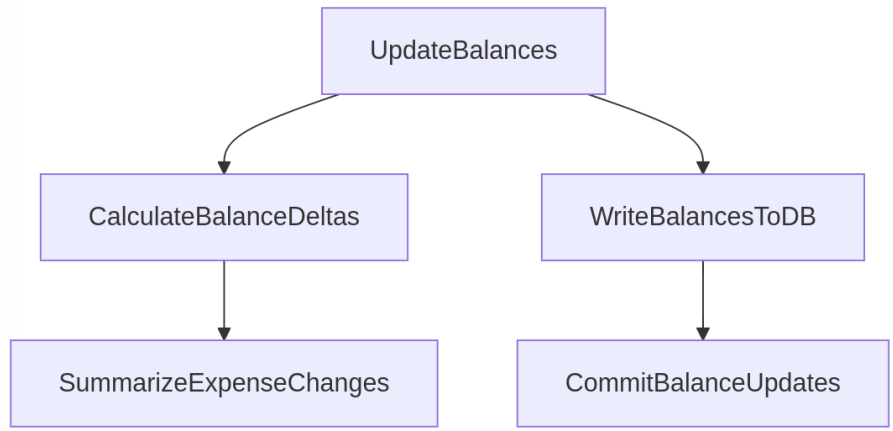
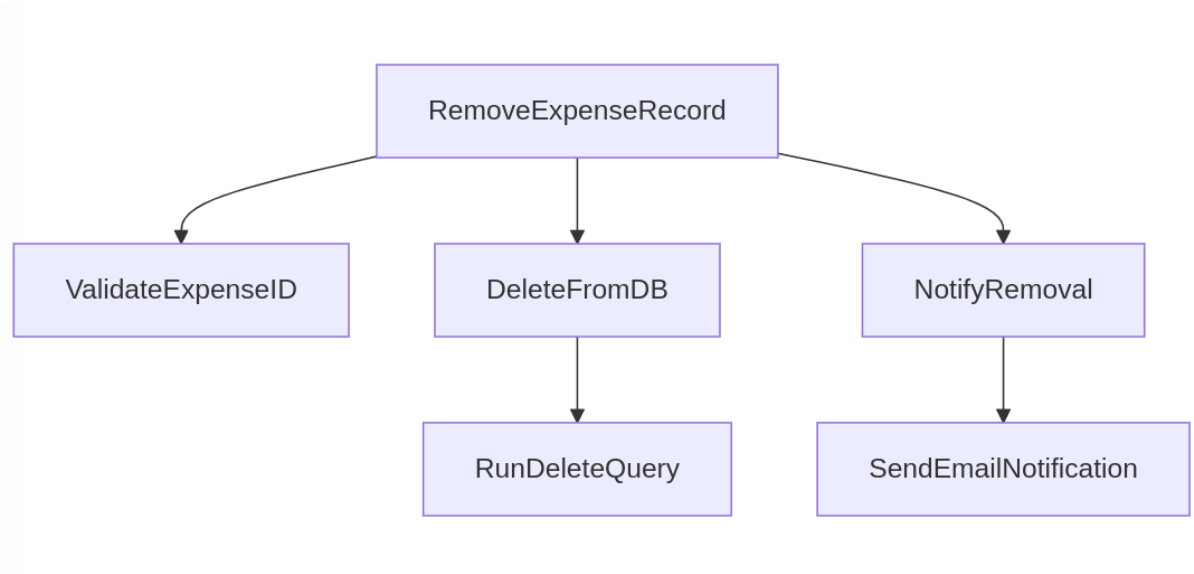
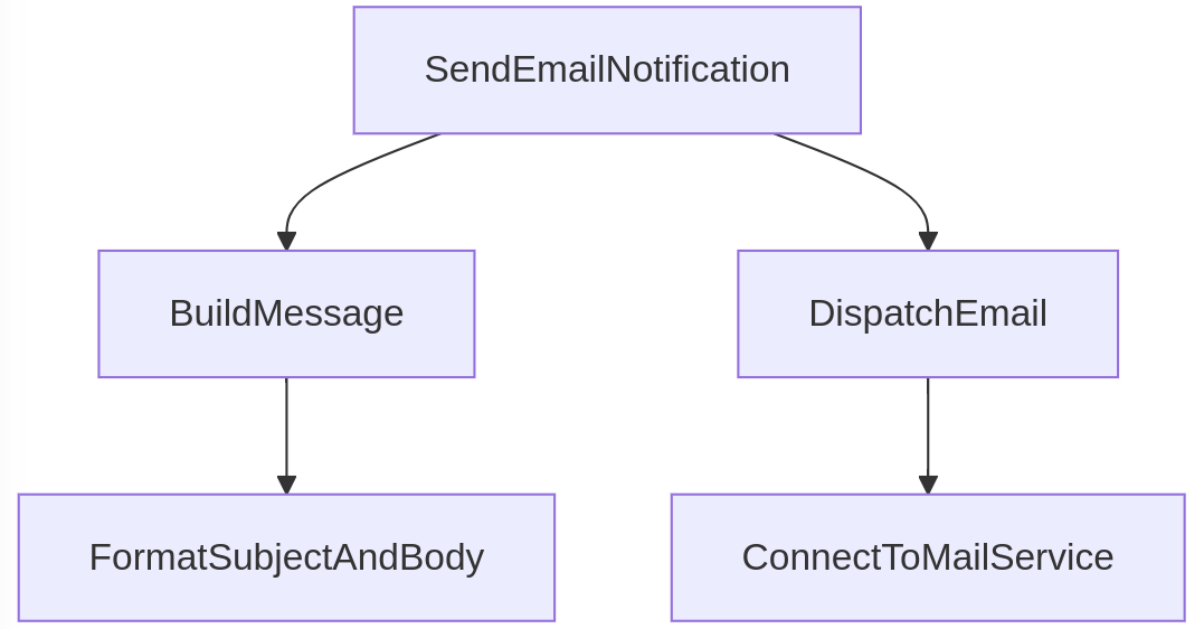


Figure 3.4: Factored Output modules




RemoveExpenseRecord
ValidateExpenseID
DeleteFromDB
NotifyRemoval
RunDeleteQuery
SendEmailNotification


SendEmailNotification
BuildMessage
DispatchEmail
FormatSubjectAndBody
ConnectToMailService

3.5 Final Structure Chart

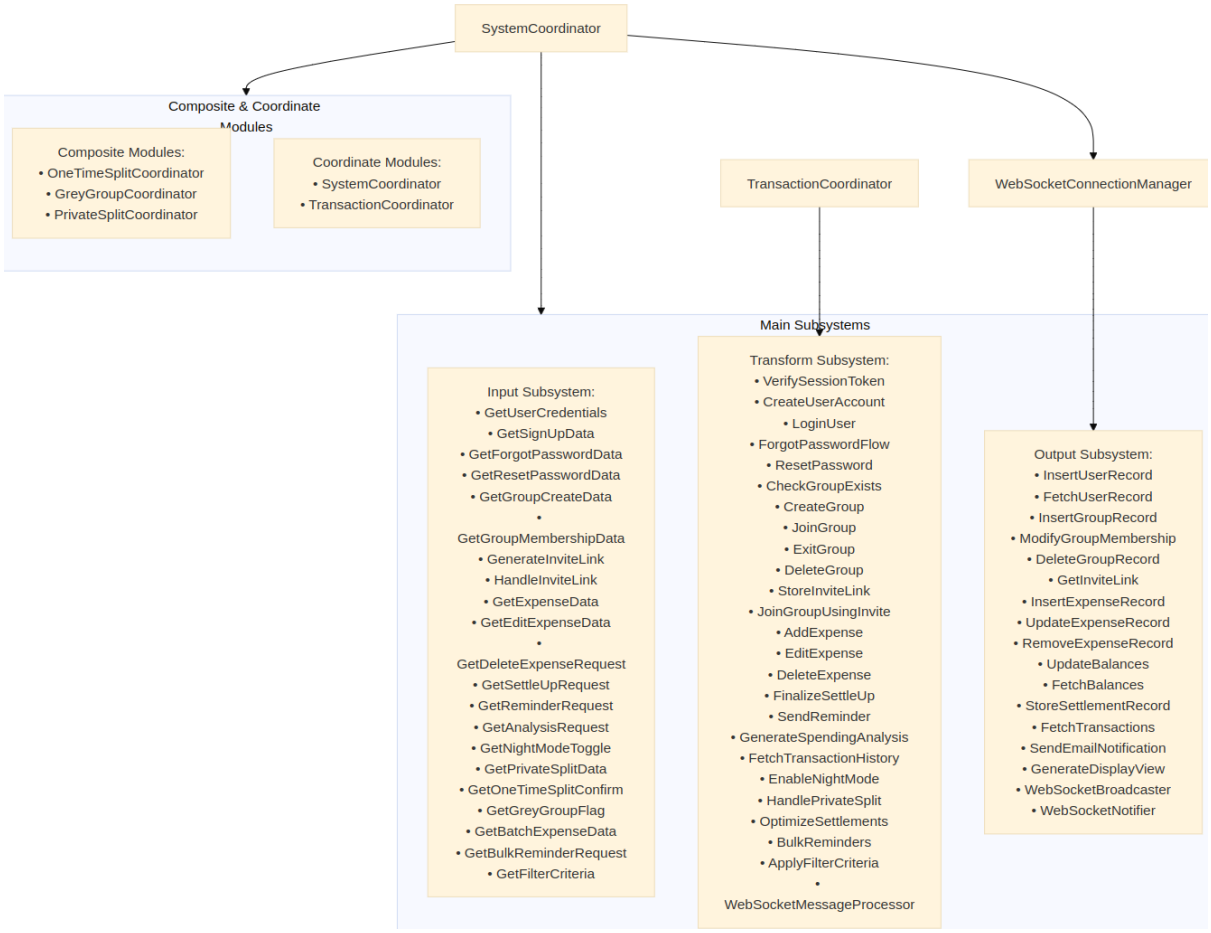


Figure 3.5: Final Structure Chart

Chapter 4

Design Analysis

4.1 List of All Final-Level Modules

The table below shows each final-level module, its type, and its cohesion. After the table, we provide short paragraphs describing each module's purpose, cohesion rationale, and coupling details.

Module Name	Type	Cohesion Type
Input Subsystem (20)		
GetUserCredentials	Input	Functional
GetSignUpData	Input	Functional
GetForgotPasswordData	Input	Functional
GetResetPasswordData	Input	Functional
GetGroupCreateData	Input	Functional
GetGroupMembershipData	Input	Functional
GenerateInviteLink	Input	Functional
HandleInviteLink	Inout	Functional
GetExpenseData	Input	Functional
GetEditExpenseData	Input	Functional
GetDeleteExpenseRequest	Input	Functional
GetSettleUpRequest	Input	Functional
GetReminderRequest	Input	Functional
GetAnalysisRequest	Input	Functional
GetNightModeToggle	Input	Functional
GetPrivateSplitData	Input	Functional
GetOneTimeSplitConfirm	Input	Functional
GetGreyGroupFlag	Input	Functional
GetBatchExpenseData	Input	Sequential
GetBulkReminderRequest	Input	Sequential
GetFilterCriteria	Input	Functional
Transform Subsystem (25)		
VerifySessionToken	Transform	Functional
CreateUserAccount	Transform	Functional
LoginUser	Transform	Functional

Module Name	Type	Cohesion Type
ForgotPasswordFlow	Transform	Sequential
ResetPassword	Transform	Sequential
CheckGroupExists	Transform	Functional
CreateGroup	Transform	Functional
JoinGroup	Transform	Functional
ExitGroup	Transform	Functional
DeleteGroup	Transform	Sequential
StoreInviteLink	Transform	Functional
JoinGroupUsingInvite	Transform	Functional
AddExpense	Transform	Sequential
EditExpense	Transform	Sequential
DeleteExpense	Transform	Sequential
FinalizeSettleUp	Transform	Sequential
SendReminder	Transform	Functional
GenerateSpendingAnalysis	Transform	Functional
FetchTransactionHistory	Transform	Functional
EnableNightMode	Transform	Functional
HandlePrivateSplit	Transform	Sequential
OptimizeSettlements	Transform	Sequential
AddExpense	Transform	Sequential
ManageAttachments	Transform	Sequential
BulkReminders	Transform	Sequential
ApplyFilterCriteria	Transform	Functional
WebSocketMessageProcessor	Transform	Sequential
Output Subsystem (15)		
InsertUserRecord	Output	Functional
FetchUserRecord	Output	Functional
InsertGroupRecord	Output	Functional
ModifyGroupMembership	Output	Functional
DeleteGroupRecord	Output	Functional
GetInviteLink	Output	Functional
InsertExpenseRecord	Output	Functional
UpdateExpenseRecord	Output	Functional
RemoveExpenseRecord	Output	Functional
UpdateBalances	Output	Functional
FetchBalances	Output	Functional
StoreSettlementRecord	Output	Functional
FetchTransactions	Output	Functional
SendEmailNotification	Output	Functional
GenerateDisplayView	Output	Functional
WebSocketBroadcaster	Output	Sequential
WebSocketNotifier	Output	Sequential
Composite (3), Coordinate (2)		
OneTimeSplitCoordinator	Composite	Sequential

Module Name	Type	Cohesion Type
GreyGroupCoordinator	Composite	Sequential
PrivateSplitCoordinator	Composite	Sequential
SystemCoordinator	Coordinate	Functional
TransactionCoordinator	Coordinate	Functional
WebSocketConnectionManager	Coordinate	Functional

Summary of Modules:

- **Input:** 20
- **Transform:** 27
- **Output:** 17
- **Composite:** 3
- **Coordinate:** 3
- **Total:** 70

Table 4.2: Most Complex or Error-Prone Modules

Subsystem	Module	Reason for Complexity / Error-Prone
Input	GetBatchExpenseData	Handles multiple expenses in a single submission; requires robust validation, looping logic, and error handling for partial failures.
Transformation	DeleteGroup	Multi-step flow with high fan-out (removing group record, clearing memberships, recalculating balances); numerous edge cases (unsettled debts, admin checks).
Output	UpdateBalances	Critical data integrity for all participants; incorrect writes can cascade into widespread inaccuracies; often tied to real-time notifications.

4.2 Short Descriptions, Cohesion & Coupling Details

Below, we provide a brief description for each module. This covers the module's purpose, why it has its stated cohesion, and how strongly or weakly it's coupled to others.

4.2.1 Input Subsystem (20)

GetUserCredentials:

- *Purpose:* Captures the username/password from the user's login screen.
- *Cohesion:* Functional, since it has a single task: gather credentials.
- *Coupling:* Low; it passes the data forward without needing anything from other modules.

GetSignUpData:

- *Purpose:* Reads all sign-up fields (name, email, username, pass) with basic format checks.
- *Cohesion:* Functional, as it focuses on a single goal: collecting sign-up info.

- *Coupling:* Low; only the `CreateUserAccount` transform module depends on it.

GetForgotPasswordData:

- *Purpose:* Collects user's email for initiating a "forgot password" flow.
- *Cohesion:* Functional; it just does a single operation of capturing email.
- *Coupling:* Low, passing the email to `ForgotPasswordFlow`.

GetResetPasswordData:

- *Purpose:* Obtains old and new passwords so the user can reset their password.
- *Cohesion:* Functional, as it focuses solely on capturing old/new pass.
- *Coupling:* Low; used by the `ResetPassword` transform.

GetGroupCreateData:

- *Purpose:* Captures group name and type (normal, grey, OTS).
- *Cohesion:* Functional; single step: gather data for group creation.
- *Coupling:* Low, referencing `CreateGroup` for actual logic.

GetGroupMembershipData:

- *Purpose:* Reads membership details for adding/removing user(s) to a group.
- *Cohesion:* Functional, capturing membership changes in one function.
- *Coupling:* Low, eventually calls `JoinGroup` or `ExitGroup`.

GenerateInviteLink:

- *Purpose:* Generates a unique invite token for a group, sets an expiration time, and constructs the full invite link to be shared.
- *Cohesion:* Functional; focuses solely on generating and preparing invite link data.
- *Coupling:* Low; interacts minimally with token generation utilities and the `StoreInviteLink` module for database persistence.

HandleInviteLink:

- *Purpose:* Processes a clicked invite link by extracting and validating the invite token, checking its expiration, and triggering the group join process.
- *Cohesion:* Functional; dedicated to handling invite link events in a single, focused workflow.
- *Coupling:* Low; relies on token validation logic and eventually calls the `JoinGroup` module to add the user to the group.

GetExpenseData:

- *Purpose:* Receives expense details—amount, desc, who paid, who's involved.
- *Cohesion:* Functional, deals with exactly one kind of input.
- *Coupling:* Low, passes final data to `AddExpense` in transforms.

GetEditExpenseData:

- *Purpose:* Gathers updated fields for an existing expense.
- *Cohesion:* Functional, used for a single operation (edit).
- *Coupling:* Low, only `EditExpense` depends on it.

GetDeleteExpenseRequest:

- *Purpose:* Confirms user's intention to delete an expense (by ID).
- *Cohesion:* Functional, capturing a single action.
- *Coupling:* Low, used by `DeleteExpense`.

GetSettleUpRequest:

- *Purpose:* Captures a user's intent to settle a debt.
- *Cohesion:* Functional, it just picks up who's paying, who's receiving, how much.
- *Coupling:* Low, feeding `FinalizeSettleUp`.

GetReminderRequest:

- *Purpose:* Reads which user or group needs a settlement reminder.
- *Cohesion:* Functional, a single action: gather reminder info.
- *Coupling:* Low, references `SendReminder`.

GetAnalysisRequest:

- *Purpose:* Specifies the analysis type, date range, or categories for analytics.
- *Cohesion:* Functional, capturing only analysis parameters.
- *Coupling:* Low, used by `GenerateSpendingAnalysis`.

GetNightModeToggle:

- *Purpose:* Obtains a boolean for toggling night mode.
- *Cohesion:* Functional, single Boolean toggle.
- *Coupling:* Low, used by `EnableNightMode`.

GetPrivateSplitData:

- *Purpose:* Collects data for a 2-person private split.

- *Cohesion:* Functional, just obtains the participants.
- *Coupling:* Low, used by `HandlePrivateSplit` or `PrivateSplitCoordinator`.

GetOneTimeSplitConfirm:

- *Purpose:* Captures user's "done adding expenses" status for OTS.
- *Cohesion:* Functional, it's a single yes/no input.
- *Coupling:* Low, triggers `OneTimeSplitCoordinator`.

GetGreyGroupFlag:

- *Purpose:* Checks if a newly created group is "Grey."
- *Cohesion:* Functional, single step.
- *Coupling:* Low, used by `CreateGroup` or `GreyGroupCoordinator`.

GetBatchExpenseData:

- *Purpose:* Reads multiple expenses in a single submission.
- *Cohesion:* Sequential, it iterates over multiple expense inputs.
- *Coupling:* Medium, because it interacts with `BatchAddExpenses` for multi-record submission.

GetBulkReminderRequest:

- *Purpose:* Accepts a list of reminder targets in one request.
- *Cohesion:* Sequential, looping to collect multiple.
- *Coupling:* Medium, interacts with `BulkReminders` transform.

GetFilterCriteria:

- *Purpose:* Specifies date range, categories, or other filters for advanced analytics.
- *Cohesion:* Functional, single job.
- *Coupling:* Low, used by `ApplyFilterCriteria` or `GenerateSpendingAnalysis`.

4.2.2 Transform Subsystem (25)

VerifySessionToken:

- *Purpose:* Checks DB for valid session token.
- *Cohesion:* Functional, it does exactly one check.
- *Coupling:* Medium, as many modules call it to confirm user identity.

CreateUserAccount:

- *Purpose:* Validates sign-up data, hashes password, calls DB insert.
- *Cohesion:* Functional, single job (create account).
- *Coupling:* Medium, depends on `InsertUserRecord` for DB writes.

LoginUser:

- *Purpose:* Verifies credentials, issues session token if correct.
- *Cohesion:* Functional, singular purpose.
- *Coupling:* Medium, calls `FetchUserRecord` and session updates.

ForgotPasswordFlow:

- *Purpose:* Generates a reset link after verifying user's email.
- *Cohesion:* Sequential, multiple steps (check DB, send token).
- *Coupling:* Medium, depends on `SendEmailNotification`.

ResetPassword:

- *Purpose:* Updates user's password given old pass or reset token.
- *Cohesion:* Sequential, verifying old pass, then updating DB.
- *Coupling:* Medium, calls `FetchUserRecord` and `InsertUserRecord`.

CheckGroupExists:

- *Purpose:* Confirms a group record is present in DB.
- *Cohesion:* Functional, one DB check.
- *Coupling:* Low, used by group-based transforms.

CreateGroup:

- *Purpose:* Creates a new group (possibly normal, grey, OTS).
- *Cohesion:* Functional, single DB insertion plus minor logic.
- *Coupling:* Medium, uses `InsertGroupRecord`.

JoinGroup:

- *Purpose:* Adds user(s) to group membership.
- *Cohesion:* Functional, straightforward membership addition.
- *Coupling:* Medium, calls `ModifyGroupMembership`.

ExitGroup:

- *Purpose:* Removes user from group membership if no debts remain.

- *Cohesion:* Functional, single flow.
- *Coupling:* Medium, checks `FetchBalances`, calls `ModifyGroupMembership`.

DeleteGroup:

- *Purpose:* Removes group record, membership references, cleans up expenses.
- *Cohesion:* Sequential, multiple sub-steps.
- *Coupling:* High, calls `DeleteGroupRecord`, `RemoveExpenseRecord`, `UpdateBalances`.

StoreInviteLink:

- *Purpose:* Stores the invite token, associated group ID, and expiration time in the database.
- *Cohesion:* Functional; it is solely focused on storing invite link data.
- *Coupling:* Low; it only interacts with the database layer and is utilized by modules like `GenerateInviteLink` to ensure invite data persistence.

JoinGroupUsingInvite:

- *Purpose:* Processes a valid invite token to add a user to the corresponding group.
- *Cohesion:* Functional; it encapsulates all steps related to converting an invite into an actual group membership.
- *Coupling:* Low; it depends on token validation and ultimately calls the `JoinGroup` module, keeping its interactions focused.

AddExpense:

- *Purpose:* Inserts new expense, updates each participant's amounts.
- *Cohesion:* Sequential, verifying membership, then DB write, then re-balancing.
- *Coupling:* High, uses `InsertExpenseRecord` and `UpdateBalances`.

EditExpense:

- *Purpose:* Modifies an existing expense and re-adjusts balances if amounts changed.
- *Cohesion:* Sequential.
- *Coupling:* High, calls `UpdateExpenseRecord`, `UpdateBalances`.

DeleteExpense:

- *Purpose:* Removes an expense record, recalculating group amounts if needed.
- *Cohesion:* Sequential, multiple steps (permission check, remove, recalc).
- *Coupling:* High, referencing `RemoveExpenseRecord`, possibly `UpdateBalances`.

FinalizeSettleUp:

- *Purpose:* Zeroes out amounts once both payer and payee confirm.
- *Cohesion:* Sequential, multiple steps (confirmation, record settlement, update).
- *Coupling:* Medium, uses `StoreSettlementRecord` and `UpdateBalances`.

SendReminder:

- *Purpose:* Dispatches a reminder for outstanding debts.
- *Cohesion:* Functional, single job.
- *Coupling:* Medium, typically calls `SendEmailNotification`.

GenerateSpendingAnalysis:

- *Purpose:* Summarizes expenses into categories/time-based stats.
- *Cohesion:* Functional, single pass for generating analysis.
- *Coupling:* Medium, depends on various queries or data fetch calls.

FetchTransactionHistory:

- *Purpose:* Retrieves personal/group transaction logs from DB.
- *Cohesion:* Functional, single operation.
- *Coupling:* Medium, calls `FetchTransactions`.

EnableNightMode:

- *Purpose:* Updates user's preference for dark mode.
- *Cohesion:* Functional.
- *Coupling:* Low, only calls a user record update routine.

HandlePrivateSplit:

- *Purpose:* Manages creation/logic for a 2-person private expense scenario.
- *Cohesion:* Sequential, verifying membership, applying privacy rules.
- *Coupling:* Medium, references `AddExpense` or membership checks.

OptimizeSettlements:

- *Purpose:* Minimizes total transaction count across group members.
- *Cohesion:* Sequential, calls `CalculateSettlement` then runs an optimization algorithm.
- *Coupling:* High, interacts with multiple settlement data flows.

AddExpenses:

- *Purpose:* Stores the expense in the database and notifies the web-socket.

- *Cohesion:* Functional; it focuses exclusively on transforming raw expense data into a final, stored expense record.
- *Coupling:* Low; it interacts with the `GetAddExpenseData` module for input and the `InsertExpenseRecord` module for output, ensuring a clear separation of concerns.

BulkReminders:

- *Purpose:* Processes a list of reminder targets in one pass.
- *Cohesion:* Sequential, it loops calling `SendReminder`.
- *Coupling:* Medium, repeated calls to `SendReminder`, plus some iteration logic.

ApplyFilterCriteria:

- *Purpose:* Filters existing expense/transaction data by date range, categories, etc.
- *Cohesion:* Functional, single function for applying filters.
- *Coupling:* Low, obtains data from DB or from prior transforms.

WebSocketMessageProcessor:

- Purpose: Processes incoming WebSocket messages and routes them accordingly.
- Cohesion: Sequential, as message handling requires validation, parsing, and delegation to appropriate handlers in a strict order.
- Coupling: Moderate; depends on WebSocketConnectionManager for client tracking and backend modules for processing specific commands.

4.2.3 Output Subsystem (15)

InsertUserRecord:

- *Purpose:* Inserts a row for a new user in the DB.
- *Cohesion:* Functional, only does an INSERT.
- *Coupling:* Low, typically invoked by `CreateUserAccount` or `ResetPassword`.

FetchUserRecord:

- *Purpose:* Retrieves user details from DB given an ID/email.
- *Cohesion:* Functional, single query.
- *Coupling:* Low, used by `LoginUser` or `ForgotPasswordFlow`.

InsertGroupRecord:

- *Purpose:* Writes a new group entry into the DB.
- *Cohesion:* Functional, single DB insertion.
- *Coupling:* Low, typically called by `CreateGroup`.

ModifyGroupMembership:

- *Purpose:* Updates membership table for user(s) join/exit.
- *Cohesion:* Functional, single responsibility.
- *Coupling:* Low, used by `JoinGroup`, `ExitGroup`.

DeleteGroupRecord:

- *Purpose:* Removes a group row from DB.
- *Cohesion:* Functional, single delete query.
- *Coupling:* Low, invoked by `DeleteGroup`.

GetInviteLink:

- *Purpose:* Retrieves the invite token and constructs the full invite link from the database for a specific group.
- *Cohesion:* Functional, dedicated solely to fetching and formatting invite link data.
- *Coupling:* Low, typically invoked by higher-level modules in the invitation workflow.

InsertExpenseRecord:

- *Purpose:* Adds a new expense entry to the DB.
- *Cohesion:* Functional, single DB insertion.
- *Coupling:* Low, used by `AddExpense` or `BatchAddExpenses`.

UpdateExpenseRecord:

- *Purpose:* Modifies an existing expense row.
- *Cohesion:* Functional, single update.
- *Coupling:* Low, called by `EditExpense`.

RemoveExpenseRecord:

- *Purpose:* Deletes an expense row from DB.
- *Cohesion:* Functional, single remove.
- *Coupling:* Low, used by `DeleteExpense`, `DeleteGroup`.

UpdateBalances:

- *Purpose:* Adjusts owes/owed amounts in a specialized balances table.
- *Cohesion:* Functional, writing new balances.
- *Coupling:* Medium, invoked by many transforms (AddExpense, EditExpense, FinalizeSettleUp).

FetchBalances:

- *Purpose:* Retrieves the current owes/owed amounts for group or user.
- *Cohesion:* Functional, single DB query.
- *Coupling:* Medium, used by `CalculateSettlement`, `ExitGroup`.

StoreSettlementRecord:

- *Purpose:* Writes a final settlement event into a settlement table.
- *Cohesion:* Functional, single insert.
- *Coupling:* Low, typically called by `FinalizeSettleUp`.

FetchTransactions:

- *Purpose:* Queries transaction logs from multiple joined DB tables.
- *Cohesion:* Functional, single read flow.
- *Coupling:* Medium, used by `FetchTransactionHistory` or analysis modules.

SendEmailNotification:

- *Purpose:* Sends an email via SMTP or a mail API.
- *Cohesion:* Functional, single job.
- *Coupling:* Medium, used by `SendReminder` or `ForgotPasswordFlow`.

GenerateDisplayView:

- *Purpose:* Assembles data from DB into a final JSON or UI display format.
- *Cohesion:* Functional, single pass at formatting.
- *Coupling:* Medium, might read from multiple queries or merges data to produce the final output.

WebSocketBroadcaster:

- Purpose: Sends real-time updates to all connected clients when relevant events occur.
- Cohesion: Sequential, as it follows a structured process: identifying recipients, formatting the message, and broadcasting it.
- Coupling: Moderate; relies on WebSocketConnectionManager for managing connections but does not depend on other logic-heavy modules.

WebSocketNotifier:

- Purpose: Sends event-based WebSocket messages to specific clients.
- Cohesion: Sequential, as it involves identifying recipients, preparing the update, and dispatching it in order.
- Coupling: Moderate; interacts with WebSocketConnectionManager and backend events but operates independently of database and other logic modules.

4.2.4 Composite (3) and Coordinate (2)

OneTimeSplitCoordinator (Composite):

- *Purpose:* Once all members confirm OTS, triggers final settlement.
- *Cohesion:* Sequential, multi-step orchestration.
- *Coupling:* High, calls `AddExpense`, `CalculateSettlement`, `UpdateBalances`.

GreyGroupCoordinator (Composite):

- *Purpose:* Filters displayed expenses in a "Grey Group" so each user sees only relevant data.
- *Cohesion:* Sequential, checking membership, limiting expense visibility.
- *Coupling:* Medium, references group membership checks, expense fetch logic.

PrivateSplitCoordinator (Composite):

- *Purpose:* Coordinates multi-step logic for a private-split creation, expense addition, or locking.
- *Cohesion:* Sequential, ensuring only two participants can see data.
- *Coupling:* Medium, calls `HandlePrivateSplit` and membership checks.

SystemCoordinator (Coordinate):

- *Purpose:* Routes user requests to input/transform/output modules.
- *Cohesion:* Functional, single role: orchestrate calls.
- *Coupling:* High, can invoke any module in the system.

TransactionCoordinator (Coordinate):

- *Purpose:* Manages advanced transaction flows, e.g. batch additions, bulk reminders, settlement optimization.
- *Cohesion:* Functional, orchestrating a single set of transaction-based flows.
- *Coupling:* High, calls `BatchAddExpenses`, `BulkReminders`, `OptimizeSettlements`.

WebSocketConnectionManager:

- Purpose: Handles WebSocket connections, tracking active clients and managing connections.
- Cohesion: Functional, as each method (connect, disconnect, fetch active clients) is an independent operation.
- Coupling: Low; interacts only with WebSocket modules and maintains a clean separation from business logic.

4.3 Top-3 Modules with Fan-In/Fan-Out

4.3.1 Top-3 Fan-In

- **VerifySessionToken** (Transform): Called by many modules requiring user auth.
- **UpdateBalances** (Output): Invoked by AddExpense, EditExpense, FinalizeSettleUp, etc.
- **SystemCoordinator** (Coordinate): The main orchestrator that routes calls (some designs exclude the coordinator from fan-in counts, but we list it here).

4.3.2 Top-3 Fan-Out

- **DeleteGroup** (Transform): Potentially calls `DeleteGroupRecord`, `RemoveExpenseRecord`, `UpdateBalances`, `ModifyGroupMembership`, etc.
- **OneTimeSplitCoordinator** (Composite): Final OTS orchestration calls multiple transform modules.
- **TransactionCoordinator** (Coordinate): Might dispatch to `BatchAddExpenses`, `BulkReminders`, `OptimizeSettlements`, etc.

Chapter 5

Detailed Design Specification

Below we show pseudo-code for **every final-level module**. The syntax is high-level and not language-specific, but it fully enumerates attributes and methods.

5.1 Input Modules (20)

- **GetUserCredentials**

```
class GetUserCredentials:
    attributes:
        username : string
        password : string

    methods:
        def capture_input() -> (string, string):
            # Step 1: Prompt user for username
            # Step 2: Prompt user for password
            # Step 3: Return (username, password)
```

- **GetSignUpData**

```
class GetSignUpData:
    attributes:
        name  : string
        email : string
        pass  : string
        username : string

    methods:
        def capture_sign_up_fields() -> (string, string, string, string):
            # Step 1: Read ‘name’ from user
            # Step 2: Read ‘email’
            # Step 3: Read ‘pass’
            # Step 4: Read ‘username’
            # Step 5: Basic format checks on email
```

```
            # Step 6: Return (name, email, pass, username)
```

- **GetForgotPasswordData**

```
class GetForgotPasswordData:
    attributes:
        email : string

    methods:
        def capture_email() -> string:
            # Step 1: Prompt user for email
            # Step 2: Validate format
            # Step 3: Return email
```

- **GetResetPasswordData**

```
class GetResetPasswordData:
    attributes:
        oldPassword : string
        newPassword : string

    methods:
        def capture_reset_info() -> (string, string):
            # Step 1: Prompt user for oldPassword
            # Step 2: Prompt user for newPassword
            # Step 3: Return them as tuple
```

- **GetGroupCreateData**

```
class GetGroupCreateData:
    attributes:
        groupName : string
        groupType : string

    methods:
        def capture_group_details() -> (string, string):
            # Step 1: Ask for groupName
            # Step 2: Ask for groupType (normal, grey, OTS)
            # Step 3: Return (groupName, groupType)
```

- **GetGroupMembershipData**

```
class GetGroupMembershipData:
    attributes:
```

```
        groupID : int
        usernames : list of string

    methods:
        def capture_membership_change() -> (int, list of string):
            # Step 1: Read groupID
            # Step 2: Ask for one or more usernames to add/remove
            # Step 3: Return (groupID, usernames)
```

- **GenerateInviteLink**

```
class GenerateInviteLink:
    attributes:
        groupID : int

    methods:
        def create_invite() -> InviteLink:
            # Step 1: Generate a unique inviteToken.
            # Step 2: Set an expiration time for the invite.
            # Step 3: Store the inviteToken, groupID, and expiresAt in
            the database.
            # Step 4: Return the full invite link.
```

- **HandleInviteClick**

```
class HandleInviteClick:
    attributes:
        inviteToken : string

    methods:
        def process_invite() -> GroupJoinStatus:
            # Step 1: Extract inviteToken from the clicked link.
            # Step 2: Validate inviteToken against the database.
            # Step 3: Check if the inviteToken is expired or revoked.
            # Step 4: If valid, fetch the corresponding groupID.
            # Step 5: call JoinGroup.
            # Step 6: Return success or failure response.
```

- **GetExpenseData**

```
class GetExpenseData:
    attributes:
        groupID : int
        amount      : float
        description : string
```

```
        username : string
        paidBy      : dict[str, float]  # Dictionary of people who paid,
        and how much they paid
        involved    : dict[str, float]  # Dictionary of people who owe,
        and how much they owe
        tag: string

    methods:
        def capture_expense() -> ExpenseInputData:
            # Step 1: read groupId
            # Step 2: read amount and description
            # Step 3: get the username of the user adding the expense
            # Step 4: read who paid (paidBy)
            # Step 5: read who is involved
            # Step 6: read expense tag
            # Step 7: assemble and return an ExpenseInputData structure
```

- **GetEditExpenseData**

```
class GetEditExpenseData:
    attributes:
        expenseID     : int
        newAmount     : float
        newDesc       : string
        newPaidBy : dict[str, float]
        newInvolved   : dict[str, float]
        newTag : string

    methods:
        def capture_edit() -> ExpenseEditData:
            # Step 1: read expenseID
            # Step 2: read newAmount, newDesc, newPaidBy, newInvolved,
            newTag
            # Step 3: Return an ExpenseEditData object
```

- **GetDeleteExpenseRequest**

```
class GetDeleteExpenseRequest:
    attributes:
        expenseID : int

    methods:
        def confirm_delete() -> int:
            # Step 1: Receive the expenseId from the frontend
            # Step 2: Assume frontend handles user confirmation
            # Step 3: return expenseId
```

- **GetSettleUpRequest**

```
class GetSettleUpRequest:
    attributes:
        username  : string
        targetUser  : string
        groupID : int

    methods:
        def capture_settle_info() -> SettlementRequest:
            # Step 1: read username
            # Step 2: read targetUser
            # Step 3: read groupID. groupID can be None in case of
            settlement in all groups.
            # Step 4: return a SettlementRequest structure
```

- **GetReminderRequest**

```
class GetReminderRequest:
    attributes:
        targetUser : string
        groupID    : int

    methods:
        def capture_reminder_info() -> ReminderRequest:
            # Step 1: read who needs reminding (targetUser)
            # Step 2: read groupID (if NULL then, send a reminder on behalf
            of all groups/ private-splits)
            # Step 3: return a ReminderRequest
```

- **GetAnalysisRequest**

```
class GetAnalysisRequest:
    attributes:
        analysisType : string
        dateRange    : (Date, Date)

    methods:
        def capture_analysis_params() -> AnalysisParams:
            # Step 1: read analysisType (spend summary, category-based, etc)
            # Step 2: optionally read start and end dates
            # Step 3: return an AnalysisParams object
```

- **GetNightModeToggle**

```
class GetNightModeToggle:
    attributes:
        nightMode : bool

    methods:
        def capture_toggle() -> bool:
            # Step 1: Detect if the user toggles the Night Mode button
            (On/Off)
            # Step 2: Return the updated Night Mode status
```

- **GetPrivateSplitData**

```
class GetPrivateSplitData:
    attributes:
        userA : string
        userB : string

    methods:
        def capture_private_split_info() -> (string, string):
            # Step 1: read two participants
            # Step 2: return them
```

- **GetOneTimeSplitConfirm**

```
class GetOneTimeSplitConfirm:
    attributes:
        username  : string
        groupID : int

    methods:
        def confirm_ots_expenses() -> OTSConfirm:
            # Step 1: read username
            # Step 2: read groupID
            # Step 3: Assume that frontend has confirmed using a dialog box
            # Step 4: return OTSConfirm
```

- **GetGreyGroupFlag**

```
class GetGreyGroupFlag:
    attributes:
        isGrey : bool

    methods:
        def capture_is_grey() -> bool:
            # Step 1: ask user "Is this a Grey Group? (yes/no)"
            # Step 2: return True if yes, else False
```

- **GetBatchExpenseData**

```
class GetBatchExpenseData:
    attributes:
        groupID          : int

    methods:
        def capture_batch_expense_data() -> BatchExpenseData:
            # Step 1: Read the groupID from the input.
            # Step 2: Retrieve all expense entries related to the group.
            # Step 3: Aggregate these expense entries into a
            BatchExpenseData object.
            # Step 4: Return the BatchExpenseData object.
```

- **GetBulkReminderRequest**

```
class GetBulkReminderRequest:
    attributes:
        groupID: int
        username: string
    methods:
        def capture_bulk_reminder_info() -> BulkReminderRequest:
            # Step 1: read the groupID and find all the members who owe
            money to the specified user.
            # Step 2: Assemble these details into a BulkReminderRequest
            object.
            # Step 3: Return the BulkReminderRequest object.
```

- **GetFilterCriteria**

```
class GetFilterCriteria:
    attributes:
        startDate : Date
        endDate   : Date
        categories: list of string

    methods:
        def capture_filters() -> FilterCriteria:
            # Step 1: read date range
            # Step 2: read categories
            # Step 3: return FilterCriteria
```

5.2 Transform Modules (25)

- **VerifySessionToken**

```
class VerifySessionToken:
    attributes:
        token : string

    methods:
        def validate() -> VerifiedUserID:
            # Step 1: Check DB for token
            # Step 2: If valid, return VerifiedUserID
            # Step 3: else raise an error
```

- **CreateUserAccount**

```
class CreateUserAccount:
    attributes:
        signUpData : SignUpData

    methods:
        def create_account() -> None:
            # Step 1: Validate signUpData
            # Step 2: Hash password
            # Step 3: Call InsertUserRecord
```

- **LoginUser**

```
class LoginUser:
    attributes:
        credentials : (string, string)

    methods:
        def login() -> SessionToken:
            # Step 1: Use FetchUserRecord with credentials
            # Step 2: If pass matches, generate a session token
            # Step 3: Return SessionToken
```

- **ForgotPasswordFlow**

```
class ForgotPasswordFlow:
    attributes:
        userEmail : string

    methods:
        def process_forgot_password() -> None:
            # Step 1: check if email is in DB
            # Step 2: generate reset token
            # Step 3: call SendEmailNotification to deliver reset link
```

- **ResetPassword**

```
class ResetPassword:
    attributes:
        oldPass : string
        newPass : string
        username  : string

    methods:
        def reset_user_password() -> None:
            # Step 1: verify oldPass or token
            # Step 2: update user record with newPass
```

- **CheckGroupExists**

```
class CheckGroupExists:
    attributes:
        groupID : int

    methods:
        def verify() -> bool:
            # Step 1: query DB for group
            # Step 2: return true if found, else false
```

- **CreateGroup**

```
class CreateGroup:
    attributes:
        groupData

    methods:
        def create_new_group() -> GroupID:
            # Step 1: get group data from GetGroupCreateData
            # Step 1: call InsertGroupRecord
            # Step 2: return the new GroupID
```

- **JoinGroup**

```
class JoinGroup:
    attributes:
        groupID : int
        usernames : list of string

    methods:
        def join() -> None:
```

```
            # Step 1: check group existence
            # Step 2: call ModifyGroupMembership to add users
```

- **ExitGroup**

```
class ExitGroup:
    attributes:
        groupID : int
        username  : string

    methods:
        def exit() -> None:
            # Step 1: check no outstanding balances for user
            # Step 2: call ModifyGroupMembership to remove username
```

- **DeleteGroup**

```
class DeleteGroup:
    attributes:
        groupID : int
        username : string

    methods:
        def remove_group() -> string: # sucess or failure message
            # Step 1: confirm that there are no unsettled debts in the
            group and that the user deleting the group is its admin
            # Step 2: if both condition are satisfied then call
            DeleteGroupRecord and RemoveExpenseRecord and return success
            message
            # Step 3: else, return error message
```

- **StoreInviteLink**

```
class StoreInviteLink:
    attributes:
        groupID     : int
        inviteToken : string
        expiresAt   : datetime

    methods:
        def save_to_database() -> bool:
            # Step 1: Insert inviteToken, groupID, and expiresAt into the
            database.
```

- **JoinGroupUsingInvite**

```
class JoinGroupUsingInvite:
    attributes:
        username      : string
        groupID     : int
        inviteToken : string

    methods:
        def join_group() -> bool:
            # Step 1: Verify the user is logged in.
            # Step 2: Check if the inviteToken is still valid.
            # Step 3: call JoinGroup
```

- **AddExpense**

```
class AddExpense:
    attributes:
        expenseData : ExpenseInputData

    methods:
        def add() -> ExpenseID:
            # Step 1: verify group membership
            # Step 2: call InsertExpenseRecord
            # Step 3: call UpdateBalances
            # Step 4: call WebSocketBroadcaster
            # Step 5: return ExpenseID
```

- **EditExpense**

```
class EditExpense:
    attributes:
        editData : ExpenseEditData

    methods:
        def edit() -> None:
            # Step 1: call UpdateExpenseRecord
            # Step 2: call UpdateBalances if the amounts changed
            # Step 3: call WebSocketBroadcaster
```

- **DeleteExpense**

```
class DeleteExpense:
    attributes:
        expenseID : string

    methods:
```

```
        def remove_expense() -> None:
            # Step 1: call RemoveExpenseRecord
            # Step 2: recalculate balances
            # Step 3: WebSocketBroadcaster
```

- **FinalizeSettleUp**

```
class FinalizeSettleUp:
    attributes:
        settleRequest : SettlementRequest

    methods:
        def finalize() -> None:
            # Step 1: check confirmations from payer & payee
            # Step 2: call StoreSettlementRecord
            # Step 3: call UpdateBalances to zero out relevant amounts
```

- **SendReminder**

```
class SendReminder:
    attributes:
        reminderReq : ReminderRequest

    methods:
        def send() -> None:
            # Step 1: figure out who needs reminding
            # Step 2: call SendEmailNotification
```

- **GenerateSpendingAnalysis**

```
class GenerateSpendingAnalysis:
    attributes:
        analysisParams : AnalysisParams

    methods:
        def generate() -> AnalysisResults:
            # Step 1: gather data from multiple expense queries
            # Step 2: compute totals, breakdown by category
            # Step 3: return an AnalysisResults object
```

- **FetchTransactionHistory**

```
class FetchTransactionHistory:
    attributes:
```

```
        GroupID : string
        dateRange     : (Date, Date)

    methods:
        def fetch() -> list of TransactionRecord:
            # Step 1: call FetchTransactions with filters
            # Step 2: return the resulting list
```

- **EnableNightMode**

```
class EnableNightMode:
    attributes:
        username   : string
        nightOn  : bool

    methods:
        def apply() -> None:
            # Step 1: update user preference
            # Step 2: change user preference in the database
```

- **HandlePrivateSplit**

```
class HandlePrivateSplit:
    attributes:
        userA : string
        userB : string

    methods:
        def create_private_split() -> None:
            # Step 1: ensure only userA and userB can see expenses
            # Step 2: set group to 'private' in DB
            # Step 3: Return groupId
```

- **OptimizeSettlements**

```
class OptimizeSettlements:
    attributes:
        groupID  : int

    methods:
        def minimize_transactions() -> list of PaymentPlan:
            # Step 1: fetch current balances
            # Step 2: run an algorithm that minimises the transactions
            and finds the new balances
            # Step 3: return that plan
```

- **AddExpense**

```
class AddExpense:
    attributes:
        expenseData : ExpenseData

    methods:
        def process_expense() -> ExpenseStatus:
            # Step 1: Add the expense in the database.
            # Step 2: call WebSocketBroadcaster
```

- **BulkReminders**

```
class BulkReminders:
    attributes:
        bulkReminderRequest : BulkReminderRequest

    methods:
        def process_bulk_reminders() -> ReminderStatus:
            # Step 1: For each username in targetUsers, verify that the
            user exists and is active.
            # Step 2: call SendEmailNotification to send the reminders.
```

- **ApplyFilterCriteria**

```
class ApplyFilterCriteria:
    attributes:
        filterCrit : FilterCriteria

    methods:
        def apply_filters() -> list of Expense:
            # Step 1: fetch data from DB
            # Step 2: filter based on date range, categories
            # Step 3: return final list
```

- **WebSocketMessageProcessor**

```
class WebSocketMessageProcessor:
   # Processes incoming WebSocket messages and routes them to appropriate
   handlers.
   methods:
   def process_message(message) -> Event:
       # Step 1: Receive and parse the raw message from a client.
       # Step 2: Validate the message format and content.
       # Step 3: Identify the appropriate handler based on message type.
       # Step 4: Dispatch the processed event to the appropriate logic modules.
```

5.3 Output Modules (15)

- **InsertUserRecord**

```
class InsertUserRecord:
    attributes:
        userObj : UserDetails

    methods:
        def insert_user() -> None:
            # Step 1: build an INSERT query
            # Step 2: run query
```

- **FetchUserRecord**

```
class FetchUserRecord:
    attributes:
        usernameOrEmail : string

    methods:
        def fetch_user() -> UserDetails:
            # Step 1: build a SELECT query
            # Step 2: run query
            # Step 3: parse results into a UserDetails object
```

- **InsertGroupRecord**

```
class InsertGroupRecord:
    attributes:
        groupName : string
        groupType : string

    methods:
        def insert_group() -> GroupID:
            # Step 1: run an INSERT for group table
            # Step 2: return new GroupID from DB
```

- **ModifyGroupMembership**

```
class ModifyGroupMembership:
    attributes:
        groupID : int
        usernames : list of string
        action  : string  # e.g. 'add' or 'remove'
```

```
    methods:
        def update_membership() -> None:
            # Step 1: for each user in userIDs
            # Step 2: run appropriate membership query (insert or delete row)
```

- **DeleteGroupRecord**

```
class DeleteGroupRecord:
    attributes:
        groupID : int

    methods:
        def delete_group() -> None:
            # Step 1: run DELETE on group table for the given groupID
```

- **GetInviteLink**

```
class GetInviteLink:
    attributes:
        groupID : int

    methods:
        def fetch_invite() -> string:
            # Step 1: Query the database for the latest valid inviteToken
            for the given groupID.
            # Step 2: If found and not expired, return the full invite link.
            # Step 3: If no valid invite is found, generate a new one
            using GenerateInviteLink.
```

- **InsertExpenseRecord**

```
class InsertExpenseRecord:
    attributes:
        expenseData : ExpenseInputData

    methods:
        def insert_expense() -> ExpenseID:
            # Step 1: build INSERT statement
            # Step 2: store in DB, get ExpenseID
            # Step 3: call SendEmailNotification
            # Step 4: return ExpenseID
```

- **UpdateExpenseRecord**

```
class UpdateExpenseRecord:
    attributes:
        expenseID  : string
        newData    : ExpenseEditData

    methods:
        def update_expense() -> None:
            # Step 1: run UPDATE for the existing expense row
            # Step 2: call SendEmailNotification
```

- **RemoveExpenseRecord**

```
class RemoveExpenseRecord:
    attributes:
        expenseID : int

    methods:
        def remove_expense_row() -> None:
            # Step 1: DELETE the row from expense table
            # Step 2: call SendEmailNotification
```

- **UpdateBalances**

```
class UpdateBalances:
    attributes:
        groupID : int
        newOwes       : dict of { userID : float }
        newOwed       : dict of { userID : float }

    methods:
        def write_balances() -> None:
            # Step 1: For each user in newOwes/newOwed
            # Step 2: update the specialized balances table
```

- **FetchBalances**

```
class FetchBalances:
    attributes:
        groupID : int

    methods:
        def get_balances() -> (dict of owes, dict of owed):
            # Step 1: SELECT from balances table
            # Step 2: return balances fetched
```

- **StoreSettlementRecord**

```
class StoreSettlementRecord:
    attributes:
        settlementInfo : SettlementRequest
        finalizeTime   : DateTime

    methods:
        def store() -> None:
            # Step 1: Insert into settlement table
```

- **FetchTransactions**

```
class FetchTransactions:
    attributes:
        groupID : int
        dateRange     : (Date, Date)

    methods:
        def fetch_all() -> list of TransactionRecord:
            # Step 1: run SELECT across joined tables
            # Step 2: filter by groupID, dateRange
            # Step 3: return a list of TransactionRecord
```

- **SendEmailNotification**

```
class SendEmailNotification:
    attributes:
        recipients : list of string
        subject    : string
        body       : string

    methods:
        def send() -> None:
            # Step 1: connect to SMTP or email service
            # Step 2: send message to all recipients
```

- **GenerateDisplayView**

```
class GenerateDisplayView:
    attributes:
        dataToFormat : any type

    methods:
        def build_view() -> string:
```

```
            # Step 1: combine data from DB queries
            # Step 2: produce a JSON representation
            # Step 3: return the formatted output
```

- **WebSocketBroadcaster**

```
class WebSocketBroadcaster: # Broadcasts real-time updates to all or a subset
   of connected clients.
   methods:
   def broadcast_message(event) -> None:
       # Step 1: Identify target clients or client groups based on event data.
       # Step 2: Format the event into a message suitable for WebSocket
       transmission.
       # Step 3: Send the formatted message to all designated clients.
```

- **WebSocketNotifier**

```
class WebSocketNotifier:
# Sends real-time updates to specific clients based on backend events.
methods:
    def send_update(update) -> None:
        # Step 1: Identify the target client(s) from the update details.
        # Step 2: Validate and format the update content.
        # Step 3: Deliver the update to the specific client(s) via WebSocket.
```

5.4 Composite Modules (3)

- **OneTimeSplitCoordinator**

```
class OneTimeSplitCoordinator:
    attributes:
        groupID : int
        confirmedUsers : set of string

    methods:
        def confirm_expenses(userID) -> None:
            # Mark userID as 'done adding expenses'
            # If all group members confirmed, call finalize_ots()

        def finalize_ots() -> None:
            # Step 1: call CalculateSettlement
            # Step 2: call UpdateBalances
            # Step 3: lock the group from further expense additions
```

- **GreyGroupCoordinator**

```
class GreyGroupCoordinator:
    attributes:
        groupID : int

    methods:
        def get_expenses_for_user(userID) -> list of Expense:
            # Step 1: fetch expenses for this group
            # Step 2: filter to only those involving userID
            # Step 3: return filtered list
```

- **PrivateSplitCoordinator**

```
class PrivateSplitCoordinator:
    attributes:
        userA : string
        userB : string
        groupID : int

    methods:
        def setup_private_split() -> None:
            # Step 1: ensure group is locked to userA and userB only
            # Step 2: call HandlePrivateSplit
```

5.5 Coordinate Modules (2)

- **SystemCoordinator**

```
class SystemCoordinator:
    # The main orchestrator that routes typical user requests

    methods:
        def handle_request(request) -> Response:
            # Step 1: parse request to identify type
            # Step 2: call relevant Input module to get data
            # Step 3: call transform modules
            # Step 4: call output modules
            # Step 5: return final result (via GenerateDisplayView)
```

- **TransactionCoordinator**

```
class TransactionCoordinator:
    # Manages multi-step or batch-based transaction flows

    methods:
        def handle_transactions(req) -> Response:
```

```
            # Step 1: call the corresponding transaction module as
            required
            # Step 2: return final result
```

- **WebSocketConnectionManager**

```
class WebSocketConnectionManager:
    # Manages the lifecycle of WebSocket connections.
    methods:
    def manage_connections() -> None:
        # Step 1: Accept incoming connection requests from clients.
        # Step 2: Maintain and update a list of active WebSocket connections.
        # Step 3: Monitor connection health and handle reconnections if
        necessary.
        # Step 4: Terminate connections when clients disconnect or on errors.
```