# Currency Exchange Logic Guide

## Initialization Sequence

### State Bootstrapping

src/main.cpp (lines 13-92) orchestrates startup. DataStore::initialize creates data/ folders, then loadRates and loadReserve either return existing CSV contents or fall back to defaults defined in defaultReserveBalances and ensureDefaultRates at src/main.cpp (lines 13-76).

If loadRates returns empty, ensureDefaultRates seeds USD, EUR, and GBP against LOCAL and writes them back via DataStore::saveRates. The same pattern seeds reserve balances and critical minimums before ConsoleUI::run is called.

### Object Wiring

RateTable and Reserve instances are passed into ExchangeOffice (constructor in src/exchange\_manager.cpp lines 139-156). ExchangeOffice clones the initial reserve to startingReserve so daily reports capture the opening snapshot.

ConsoleUI (src/console\_ui.cpp lines 1-356) receives references to ExchangeOffice and DataStore, so every user action directly mutates the live office state and immediately persists changes.

## Transaction Processing Logic

### Request Validation

ExchangeRequest enforces invariants in src/utils.cpp (lines 67-112). It rejects non-positive totals, more than one remainder portion, or allocations exceeding totalAmount. This guards ExchangeOffice::executeTransaction from malformed input.

### Execution Flow

ExchangeOffice::executeTransaction (src/exchange\_manager.cpp lines 157-246) iterates over requested ExchangePortion entries. For each portion it determines the source slice (either explicit sourceAmount or remaining balance when useRemainder is true).

RateTable::convert (src/exchange\_manager.cpp lines 69-108) supplies conversion rates. The method first checks for direct rate pairs, then falls back to the base currency (Currency::LOCAL by default). Missing rates trigger RateNotFoundError, bubbling up to the UI.

Commission is calculated as commissionPercent \* convertedAmount (commissionFor helper). The office withdraws the full converted amount from the target currency reserve, re-deposits the commission, and credits the source currency to reflect received funds.

Profit accrues in base currency by converting the commission amount via RateTable::convert, ensuring the profit ledger is comparable across currencies. Each processed portion reduces remainingSource to stop over-withdrawal.

After processing portions, ExchangeOffice records a TransactionRecord with profit, cashier details, and timestamp. The matching Receipt embeds payout breakdown, commissions, and denominations for printing.

## Reserve and Alert Logic

Reserve::withdraw checks availability using canWithdraw with an epsilon guard to avoid floating point glitches (src/exchange\_manager.cpp lines 17-41). Negative deposits or withdrawals trigger ReserveError.

ExchangeOffice::isBelowCritical compares Reserve balances against criticalMinimums populated during startup or manager adjustments. Cashier::collectLowReserveAlerts (src/employee.cpp lines 38-59) aggregates currencies below threshold for console warnings.

## Persistence Lifecycle

DataStore centralizes filesystem I/O in src/persistence.cpp (lines 1-247). CSV files store reserves (reserve.csv), rates (rates.csv), and critical minima (critical.csv).

People registry in people.csv maps canonicalized role-name keys to stable IDs. ensurePersonId trims whitespace, uppercases the role portion, and persists immediately so repeated logins reuse identifiers.

appendTransaction streams receipts into transactions.log using pipe-separated values. Each call is triggered from ConsoleUI::employeeExchangeFlow right after a successful cashier operation.

persistReport writes timestamped summaries into data/reports/report-YYYYMMDD-HHMMSS.txt, enumerating balances, thresholds, and per-transaction profit, giving managers a durable audit trail.

## Manager Operations

Manager::setExchangeRate delegates to ExchangeOffice::updateRate, which wraps RateTable::setRate. Because setRate stores inverse rates automatically (src/exchange\_manager.cpp lines 45-68), future conversions stay consistent.

Manager::setCriticalReserve updates criticalMinimums through ExchangeOffice::setCriticalMinimum and persists via DataStore::saveCriticalMinimums. ConsoleUI ensures persistence happens after every change (src/console\_ui.cpp lines 310-323).

Manager::compileDailyReport simply proxies ExchangeOffice::compileDailyReport, which packages starting balances, current balances, transaction history, and accumulated profit for the current cycle.

## Daily Cycle Reset

When a manager chooses reset, ExchangeOffice::resetDailyCycle (src/exchange\_manager.cpp lines 363-374) copies current reserve balances into startingReserve, clears dailyTransactions, and zeroes accumulated profit. ConsoleUI persists the reserve snapshot immediately afterward.

## Error Handling and User Feedback

Custom exception types (ExchangeError, RateNotFoundError, ReserveError in src/utils.cpp lines 34-53) give semantic meaning to failure modes. ConsoleUI catches std::exception in both cashier and manager flows, echoing the message to the user.

Input helpers loop until valid data is provided, preventing propagation of invalid currencies, numbers, or yes/no responses. This defensive UX keeps the domain layer free from low-level validation concerns.

## Testing and First-Run Expectations

tests/test\_basic.sh (lines 1-24) scripts a cashier session to ensure the binary can be driven non-interactively. It wipes data/ beforehand to simulate a pristine environment, mirroring the initialization path described above.

Successful execution yields a printed receipt, confirms the cashier handling record, and shows the shutdown banner, proving the main happy-path is stable.

## Extensibility Notes

Because ExchangeOffice encapsulates all financial logic, new front ends (e.g., REST API, GUI) can reuse existing flows by constructing ExchangeRequest objects and calling executeTransaction. DataStore already abstracts persistence, but swapping to a database would mean reimplementing the same interface.

BonusPolicy is polymorphic, so alternative incentive schemes can be injected by supplying a different implementation when constructing Manager instances.