MGX (Mastercard Game Exchange)
System Design Document

1. Executive Summary

MGX is a local-deployment platform that enables users to convert bank reward points into MGC (Mastercard Game Coin), store MGC in a wallet, and spend MGC to acquire UGC (Unique Game Coin) for specific games. Each UGC purchase creates a developer receivable (unsettled amount owed). Game developers can trigger settlement on demand, which batches and pays all outstanding receivables. Foreign exchange (FX) rates are cached and refreshed twice daily; each receivable stores the FX window/snapshot used to price the amount due.

2. Scope, Goals, and Non-goals

2.1 Goals

- Correct, auditable accounting for Reward Points, MGC, and per-game UGC balances (ledger-based).
- Fast synchronous user flows (top-up and purchase) with strong consistency.
- Developer-triggered, batch settlement (pull model) with safe reservation to prevent double payouts.
- FX caching with rate windows refreshed twice per day; deterministic settlement amounts.

2.2 Non-goals (Internship Scope)

- Real bank transfers or payment rails (implemented as a local Bank Adapter simulator).
- Real game-provider wallet integrations (implemented as a local Game Adapter simulator).
- KYC/AML/PCI compliance requirements.

3. High-Level Architecture

The system is designed as a modular Spring Boot backend plus a Next.js frontend. Infrastructure components (PostgreSQL, Redis, and optionally RabbitMQ) are self-hosted locally.

3.1 Services and Ports

Component	Purpose	Port	Notes
Next.js Web App	User/Developer/Admin UI	3000	Browser-facing UI
MGX API (Spring Boot)	Business logic + APIs	8081	JWT auth, ledger, settlement orchestration

PostgreSQL	Primary transactional store	5432	Recommended for locking & consistency
Redis	Cache + idempotency store	6379	FX windows, active rates, idempotency keys
RabbitMQ (optional)	Async queue for settlement workers	5672	Can be replaced with Redis Streams
FX Local Service	Provides FX quotes	8090	Mock from file or fetch-from-internet job
Bank Adapter Simulator	Simulated payouts to developer	8091	Used by settlement worker
Game Adapter Simulator (optional)	Simulated crediting of UGC to game	8092	Useful for demo realism

4. Tech Stack

4.1 Frontend

- Next.js (App Router) for UI: user wallet, top-up, purchase, history, developer settlement dashboard, admin rate management.
- API calls from browser to Spring Boot on port 8081.

4.2 Backend

- Spring Boot 3.x modular monolith (packages/modules for Auth, Wallet/Ledger, Rates, FX, Purchases, Settlements, Reporting).
- Spring Security with JWT-based authentication and role-based access control (USER, DEVELOPER, ADMIN).
- Spring Data JPA + Flyway (or Liquibase) migrations.
- Resilience4j for retries/circuit breakers when calling local adapters (FX service, Bank adapter).
- Scheduler (@Scheduled) for twice-daily FX refresh.

4.3 Data and Infrastructure

- PostgreSQL as the source of truth (strong transactions, row locking, indexing).
- Redis for hot caches (FX windows, active manual rates), idempotency keys, and optional lightweight queues.
- RabbitMQ (optional) for asynchronous settlement execution and retries.

5. Core Domain Concepts

Actors

- User: holds reward points, MGC, and per-game UGC balances.
- Developer: owns one or more games; can view receivables and request settlements.
- Admin: configures games, developers, and manual rates.

Assets

- Reward Points: user's bank loyalty points (internal balance for prototype).
- MGC: Mastercard Game Coin, stored in a user MGC wallet.
- UGC: Unique Game Coin for each game; stored per game in the user's UGC wallet.

6. Data Model (PostgreSQL)

The system uses a ledger-based accounting model. Wallet balances provide fast reads, while ledger entries provide an immutable audit trail. All money-changing operations are executed within a single database transaction.

6.1 Key Tables

Table	Purpose	Notes / Key Fields
wallets	Balances for points/MGC/UGC	type={REWARD_POINTS,MGC,UGC}, game_id for UGC, balance, version
ledger_entries	Append-only audit trail	ref_type/ref_id, wallet_id, direction, asset_type, amount, created_at
rate_points_mgc	Manual points→MGC rates	points_per_mgc, active_from/to, created_by
rate_mgc_ugc	Manual MGC→UGC rates per game	ugc_per_mgc, game_id, active_from/to
topups	Points→MGC transactions	idempotency_key, snapshots of rate used, status
purchases	MGC→UGC transactions	idempotency_key, snapshots of rate used, status
fx_rate_windows	FX batches refreshed 2x/day	valid_from/to, provider, status

Table	Purpose	Notes / Key Fields
fx_rates	USD-base quotes per window	1 USD = rate quote_currency
receivables	Unsettled amounts owed to developers	status={UNSETTLED,RESERVED,SETTLED}, amount_due, currency, fx snapshot
settlement_batches	Developer-triggered settlement requests	status={REQUESTED,PROCESSING,PAID,FAILED}, totals, timestamps

7. FX System (Cached, Refreshed Twice Daily)

FX rates are maintained as rate windows. The backend refreshes FX quotes twice per day into a new window, stores them in PostgreSQL, and caches the active window in Redis for fast access. Receivables store the FX window and computed rate used, ensuring deterministic settlement.

7.1 Optimised FX Representation (USD Base)

Store only USD→X quotes per window (e.g., USD→INR, USD→EUR). To convert A→B, compute: rate(A→B) = (USD→B) / (USD→A). This avoids maintaining N^2 pair rates.

7.2 Refresh Job (2x/day)

- At scheduled times (e.g., 00:00 and 12:00 IST), call the local FX service for USD-base quotes.
- Write a new fx_rate_window row (valid_from=now, valid_to=next refresh time) and associated fx_rates rows.
- Update Redis keys: fx:window:current = <window_id> and fx:rates:<window_id> = {quote_currency → rate}.
- If refresh fails, keep last successful window up to a configured maximum staleness and mark refresh status for visibility.

8. Caching Strategy (Redis)

- FX window and rates: fx:window:current, fx:rates:<window_id>.
- Active manual rates: rate:points_mgc:active and rate:mgc_ugc:<gameId>:active (short TTL).
- Idempotency keys: idempo:topup:<userId>:<key> and idempo:purchase:<userId>:<key>.
- Optional developer dashboard aggregates: dev:unsettled_total:<developerId> (short TTL).

Do not cache wallet balances unless strict invalidation is implemented; rely on PostgreSQL as the source of truth.

9. API Design

The frontend calls the backend API directly. Authentication is recommended via JWT in the Authorisation header to avoid cross-origin cookie complexity.

9.1 Authentication and Roles

- Roles: USER, DEVELOPER, ADMIN.
- JWT is issued on login; the frontend attaches the JWT as Authorisation: Bearer <token>.
- CORS enabled on backend for the origin, allowing Authorisation and Idempotency-Key headers.

9.2 Key Endpoints

User

- GET /v1/wallets
- POST /v1/wallets/mgc/topup (Idempotency-Key required)
- POST /v1/games/{gameId}/purchase (Idempotency-Key required)
- GET /v1/transactions/topups
- GET /v1/transactions/purchases

Developer

- GET /v1/developer/receivables?status=UNSETTLED
- POST /v1/developer/settlements/request
- GET /v1/developer/settlements/{batchId}
- GET /v1/developer/settlements

Admin

- POST /v1/admin/rates/points-mgc
- POST /v1/admin/rates/mgc-ugc
- POST /v1/admin/fx/refresh (manual trigger for testing)
- POST /v1/admin/games

10. Core Workflows

10.1 Top-up: Reward Points → MGC (Stored in Wallet)

1. Require Idempotency-Key. If key already processed, return the original result.

2. Load active points→MGC rate (Redis cache, fallback to DB).

3. Validate user has sufficient reward points.

4. Within a single DB transaction: create topup record, write ledger entries (DEBIT points, CREDIT MGC), update wallet balances, mark topup COMPLETED.

5. Persist the idempotency mapping (Redis and/or DB uniqueness).

10.2 Purchase: Spend MGC → Receive UGC + Create Developer Receivable

1. Require Idempotency-Key. If key already processed, return the original result.

2. Load active MGC→UGC rate for the selected game.

3. Validate user has sufficient MGC.

4. Compute UGC to credit and compute receivable amount_due in the developer settlement currency using the current FX window (if cross-currency).

5. Within a single DB transaction: create purchase record, write ledger entries (DEBIT MGC, CREDIT UGC), create receivable (UNSETTLED) including FX snapshot fields, mark purchase COMPLETED.

10.3 Developer Settlement (On-demand, Batch Pull)

1. Developer requests settlement via POST /v1/developer/settlements/request.

2. Create settlement batch (REQUESTED).

3. Reserve receivables atomically: select receivables where status=UNSETTLED and developer_id matches; lock and mark them RESERVED, attach settlement_batch_id. In PostgreSQL this is done safely with SELECT ... FOR UPDATE SKIP LOCKED.

4. Publish a settlement job to the local queue (RabbitMQ or Redis Streams). Return the batch id immediately.

5. Worker processes the batch: mark PROCESSING, call Bank Adapter Simulator, then mark receivables SETTLED and batch PAID (or FAILED with reason).

11. Async Processing

Recommended option: RabbitMQ locally for settlement batch execution and retries.

- Queue: settlement.batch.requested
- Dead-letter queue for repeated failures: settlement.batch.dlq
- Worker can be implemented as a separate Spring Boot profile/process or as an internal @Scheduled consumer.

12. Correctness and Reliability Mechanisms

- Idempotency: required for top-ups and purchases; enforced via unique (user_id, idempotency_key) plus Redis fast-path.
- Atomicity: each money-changing operation is a single database transaction (ledger + balance updates + transaction row).

- Concurrency safety: wallet updates use optimistic locking (version field).
- Settlement safety: receivables are reserved before processing to prevent double settlement.
- Deterministic FX: receivable stores fx_window_id and fx_rate_used; settlement never recomputes FX later.

13. Security

- JWT authentication; roles enforced on each endpoint.
- CORS restricted to frontend address for development.
- Input validation: positive amounts, existence of active rates, sufficient balances.
- Audit logging: include topup_id, purchase_id, settlement_batch_id in logs.

14. Observability

- Spring Boot Actuator: /actuator/health and /actuator/metrics.
- Structured logs with correlation/request id and domain ids (topup, purchase, batch).
- Optional: run Prometheus + Grafana locally for dashboards.