Gold Brokerage Platform - Event Sourcing Architecture Documentation

Executive Summary

This document outlines the architecture for a gold brokerage platform that allows users to purchase fractional quantities of gold, similar to cryptocurrency exchanges. The platform uses Event Sourcing as its core architectural pattern to maintain a complete audit trail of all financial transactions and gold ownership changes.

---

1. System Overview

1.1 Platform Purpose

A brokerage/investment platform enabling users to:

- Purchase small quantities of gold (fractional ownership)
- Deposit funds via bank integration
- Withdraw funds to their bank accounts
- Track gold holdings and portfolio value
- View transaction history
- Monitor market prices and estimated returns
- Participate in pooled purchases before bulk gold acquisition

1.2 Core Architecture Pattern: Event Sourcing

Event Sourcing means that every state change in the system is captured as an immutable event. Instead of storing just the current state, we store the complete history of events that led to that state.

Benefits for this platform:

- **Auditability**: Complete financial audit trail required for regulatory compliance
- **Temporal Queries**: Can reconstruct user balances at any point in time
- **Debugging**: Can replay events to understand system behavior
- **Business Intelligence**: Rich historical data for analytics
- **Compliance**: Immutable record of all transactions
- **Reconciliation**: Easy to reconcile with bank statements

---

2. Event Sourcing Fundamentals

2.1 Core Concepts

Events: Immutable facts about what happened in the system

- `UserRegistered`
- `FundsDeposited`
- `GoldPurchased`
- `WithdrawalRequested`
- `MarketPriceUpdated`

Event Store: Database that stores all events in append-only manner

Aggregates: Business entities that process commands and emit events

- User Account
- Wallet
- Gold Holdings
- Transaction

Projections: Read models built from events for querying

- User Dashboard View
- Transaction History View
- Portfolio Summary View

Command: Request to perform an action

- `RegisterUser`
- `DepositFunds`
- `PurchaseGold`
- `RequestWithdrawal`

2.2 Event Flow

```
User Action → Command → Aggregate → Event(s) → Event Store → Projections → User View
```

Example: User deposits money

1. User submits deposit via API Gateway

2. Command: `DepositFunds(userId, amount, bankReference)`
3. Wallet Aggregate validates and emits: `FundsDeposited` event
4. Event is stored in Event Store with metadata (timestamp, sequence, aggregate ID)
5. Event handlers update read models (dashboard, transaction history)
6. User sees updated balance in real-time

---

3. System Architecture

3.1 Service Boundaries

API Gateway Service

- Authentication & Authorization
- Rate limiting
- Request routing
- JWT token management
- User registration/login

Bank Integration Service

- Webhook receiver for bank notifications
- Deposit verification
- Withdrawal processing
- Bank API client
- Payment reconciliation

Gold Trading Service

- Gold purchase processing
- Fractional ownership calculation
- Pooled investment management
- Gold allocation to users
- Bulk purchase coordination

Market Data Service

- Real-time gold price updates

- Historical price tracking
- Market data aggregation
- Price event emission

Portfolio Calculator Service

- Returns calculation
- Portfolio valuation
- Performance metrics
- What-if analysis

Notification Service

- Transaction confirmations
- Price alerts
- Investment updates
- Email/SMS/Push notifications

Projection Service

- Dashboard views
- Transaction history
- Holdings summary
- Market analysis views

---

4. Database Recommendations

4.1 Event Store Database

Primary Recommendation: PostgreSQL

Why PostgreSQL for Event Store:

- ACID compliance essential for financial data
- Excellent JSON/JSONB support for event payloads
- Robust indexing for temporal queries
- Mature replication and backup solutions
- Built-in partitioning for scaling event tables

- Strong consistency guarantees
- Wide operational expertise available
- Cost-effective for production

Event Store Schema Pattern:

```
Table: events
- event_id (UUID, primary key)
- aggregate_id (UUID, indexed)
- aggregate_type (string)
- event_type (string)
- event_data (JSONB)
- metadata (JSONB)
- sequence_number (bigint, indexed)
- timestamp (timestamp with timezone)
- user_id (UUID, indexed)
```

Alternative: EventStoreDB

- Purpose-built for event sourcing
- Optimistic concurrency control
- Built-in projections
- More complex to operate

4.2 Read Model Database

Recommendation: PostgreSQL (Separate Instance)

Why separate database:

- Event store optimized for writes
- Read models optimized for queries
- Independent scaling
- Projection rebuilding doesn't impact event store

Read Model Tables:

- `user_accounts` - User profile and authentication
- `user_wallets` - Current balance and wallet status
- `gold_holdings` - Current gold ownership per user
- `transaction_history` - Denormalized transaction view

- `market_prices` - Current and historical prices
- `portfolio_valuations` - Pre-calculated portfolio values

4.3 Cache Layer

Recommendation: Redis

Use cases:

- Session management
- Real-time portfolio calculations
- Market price caching
- Rate limiting counters
- Frequently accessed user data
- Distributed locks for critical operations

---

5. Event Design

5.1 Core Events

User & Account Events

```
UserRegistered
- userId
- email
- kycStatus
- registeredAt

UserVerified
- userId
- verificationType
- verifiedAt

WalletCreated
- walletId
- userId
- currency
- createdAt
```

Financial Transaction Events

FundsDeposited
- transactionId
- userId
- walletId
- amount
- currency
- bankReference
- depositedAt
- verificationStatus

DepositConfirmed
- transactionId
- bankConfirmationId
- confirmedAt

WithdrawalRequested
- transactionId
- userId
- walletId
- amount
- bankAccountDetails
- requestedAt

WithdrawalCompleted
- transactionId
- bankReference
- completedAt

WithdrawalFailed
- transactionId
- reason
- failedAt

Gold Trading Events

GoldPurchaseInitiated
- purchaseId
- userId
- walletId
- goldAmount (grams)
- pricePerGram
- totalCost
- initiatedAt

GoldPurchaseCompleted
- purchaseId
- confirmationNumber
- completedAt

GoldAllocated
- allocationId
- userId
- goldAmount
- certificateNumber
- allocatedAt

PooledInvestmentOpened
- poolId
- targetAmount
- minimumParticipants
- openedAt

PooledInvestmentClosed
- poolId
- totalParticipants
- totalAmount
- closedAt

Market Events

```
MarketPriceUpdated
- priceId
- goldPricePerGram
- currency
- source
- updatedAt

MarketOpened
- marketSession
- openedAt

MarketClosed
- marketSession
- closedAt
```

5.2 Event Metadata Standards

Every event includes:

```
metadata:
  - eventId (UUID)
  - correlationId (UUID) - Links related events
  - causationId (UUID) - The command/event that caused this event
  - timestamp (ISO 8601)
  - userId (if applicable)
  - ipAddress
  - userAgent
  - version (event schema version)
```

---

6. Implementation Flow by Feature

6.1 User Registration & Login

Flow:

1. User submits registration via API Gateway
2. API Gateway validates input
3. Command: `RegisterUser` sent to User Service
4. User Aggregate validates uniqueness
5. Event: `UserRegistered` emitted
6. Event stored in Event Store

7. Projection handler creates user record in read database
8. Event: `WalletCreated` automatically emitted
9. Confirmation sent to user

Events:

- `UserRegistered` → `WalletCreated` → `WelcomeEmailSent`

6.2 Bank Deposit Integration

Flow:

1. User initiates bank transfer using provided reference
2. Bank sends webhook notification to Bank Integration Service
3. Service validates webhook signature
4. Event: `FundsDeposited` emitted with `pending` status
5. Background verification process
6. Event: `DepositConfirmed` emitted after verification
7. Dashboard projection updated in real-time
8. User receives notification

Events:

- `FundsDeposited` → `DepositVerificationStarted` → `DepositConfirmed` → `WalletBalanceUpdated`

Idempotency Handling:

- Use bank's transaction reference as idempotency key
- Prevent duplicate processing of same deposit

6.3 Gold Purchase

Flow:

1. User views current gold price on dashboard
2. User specifies amount to purchase (in currency or grams)
3. Command: `PurchaseGold` sent to Gold Trading Service
4. Wallet Aggregate checks sufficient balance
5. Market price locked for transaction
6. Events emitted:

- `GoldPurchaseInitiated`
- `FundsReserved`

7. Gold Trading Service allocates fractional gold
8. Event: `GoldAllocated` emitted
9. Event: `GoldPurchaseCompleted` emitted
10. Portfolio projection updated
11. Transaction history updated

Events:

- `GoldPurchaseInitiated` → `FundsReserved` → `GoldAllocated` → `GoldPurchaseCompleted` → `PortfolioUpdated`

6.4 Withdrawal Processing

Flow:

1. User requests withdrawal via dashboard
2. Command: `RequestWithdrawal` validated
3. Event: `WithdrawalRequested` emitted
4. Funds locked in wallet
5. Manual/automated approval process
6. Bank Integration Service initiates bank transfer
7. Event: `WithdrawalProcessing` emitted
8. Bank confirms transfer
9. Event: `WithdrawalCompleted` emitted
10. Wallet balance updated

Events:

- `WithdrawalRequested` → `WithdrawalApproved` → `WithdrawalProcessing` → `WithdrawalCompleted`

6.5 Dashboard Updates

Projections:

User Dashboard Projection:

- Subscribes to: `FundsDeposited`, `DepositConfirmed`, `WithdrawalCompleted`, `GoldPurchaseCompleted`
- Updates: Wallet balance, available funds, reserved funds

Gold Holdings Projection:

- Subscribes to: `GoldAllocated`, `GoldSold`
- Updates: Total gold owned (in grams), average purchase price

Transaction History Projection:

- Subscribes to: All transaction events
- Creates denormalized view with all user transactions

Portfolio Value Projection:

- Subscribes to: `MarketPriceUpdated`, `GoldAllocated`
- Recalculates current value and returns

6.6 Market Data Integration

Flow:

1. Market Data Service polls gold price APIs
2. Detects price change
3. Event: `MarketPriceUpdated` emitted
4. Portfolio Calculator Service receives event
5. Recalculates all portfolio values
6. Events: `PortfolioValuationUpdated` for affected users
7. Real-time updates pushed to connected clients

Events:

- `MarketPriceUpdated` → `PortfolioValuationUpdated` → `PriceAlertTriggered` (if thresholds met)

6.7 Pooled Investment Feature

Flow:

1. Admin creates pooled investment opportunity
2. Event: `PooledInvestmentOpened` emitted
3. Users commit funds to pool
4. Events: `PoolParticipationCommitted` for each user
5. When target reached: Event `PooledInvestmentClosed`
6. Bulk gold purchase executed

7. Event: `BulkGoldPurchased`
8. Gold allocated proportionally to participants
9. Events: `GoldAllocated` for each participant

Events:

- `PooledInvestmentOpened` → `PoolParticipationCommitted` (multiple) → `PooledInvestmentClosed` → `BulkGoldPurchased` → `GoldAllocated` (multiple)

---

7. Event Store Operations

7.1 Writing Events

Command Processing:

```
1. Receive command
2. Load aggregate from event stream
3. Validate command against current state
4. Generate new event(s)
5. Append events to event store (atomic operation)
6. Publish events to event bus
7. Return success to client
```

Optimistic Concurrency:

- Each aggregate has expected version
- Write fails if version mismatch (concurrent modification detected)
- Client retries with fresh state

7.2 Reading Events

Aggregate Reconstruction:

```
1. Query events for aggregate_id ordered by sequence
2. Apply events to empty aggregate state
3. Return current state
```

Snapshotting (Performance Optimization):

- Periodically save aggregate state snapshot
- Reconstruct from snapshot + subsequent events
- Reduces event replay overhead

7.3 Event Processing

Immediate Consistency (Write Side):

- Event stored before acknowledgment
- Strong consistency guaranteed

Eventual Consistency (Read Side):

- Projections updated asynchronously
- Usually milliseconds delay
- User sees "Processing..." state if needed

---

8. Data Consistency & Recovery

8.1 Transaction Boundaries

Within Single Aggregate:

- All events for one command are atomic
- Example: Gold purchase either fully succeeds or fails

Across Aggregates:

- Use Saga pattern for distributed transactions
- Example: Withdrawal spans Wallet + Bank Integration

Saga Example - Withdrawal:

```
1. Wallet: Reserve funds (FundsReserved event)
2. Bank Service: Initiate transfer
   - Success: WithdrawalCompleted
   - Failure: FundsReservationCancelled (compensation)
```

8.2 Event Store Backup

Strategy:

- Continuous WAL archiving (PostgreSQL)
- Daily full snapshots
- Point-in-time recovery capability
- Offsite backup replication

- Test restore procedures monthly

8.3 Projection Rebuilding

When Needed:

- Bug fix in projection logic
- Schema change in read model
- Data corruption recovery
- Adding new projection

Process:

1. Create new empty projection database
2. Replay all events from beginning
3. Validate data integrity
4. Swap to new projection (blue-green deployment)

---

9. Security & Compliance

9.1 Event Encryption

Sensitive Data in Events:

- Bank account details
- Personal identification
- Financial amounts (depending on regulations)

Approach:

- Encrypt event payload at application level
- Store encryption keys in HSM or key management service
- Maintain event searchability where needed (encrypted indexes)

9.2 Audit Trail

Benefits of Event Sourcing:

- Complete audit trail built-in
- Who did what, when, and why
- Regulatory compliance (MiFID II, financial regulations)

- Dispute resolution
- Fraud detection

Audit Queries:

- "Show all transactions for user X in date range"
- "When was this gold purchased and at what price?"
- "Reconstruct account balance at specific date"

9.3 GDPR Compliance

Challenge:

- Right to be forgotten vs immutable events

Solutions:

- Store PII in separate encrypted store with references
- Crypto-shredding: Delete encryption keys to make events unreadable
- Pseudonymization: Replace identifiers in events
- Event replacement (controversial, breaks immutability)

---

10. Scaling Strategy

10.1 Horizontal Scaling

Event Store:

- Partition by aggregate_id
- Read replicas for projections
- Archive old events to cold storage

Services:

- Stateless microservices
- Load balancer distribution
- Auto-scaling based on demand

10.2 Event Bus

Recommendation: Apache Kafka or RabbitMQ

Kafka Benefits:

- High throughput
- Event replay capability
- Partitioned for parallel processing
- Persistent message store

Event Publishing Pattern:

```
Event Store → Event Bus → Multiple Consumers (Projections, Notifications, Analytics)
```

10.3 Caching Strategy

Cache Invalidation:

- Subscribe to relevant events
- Update cache when events occur
- Use cache-aside pattern for rare queries

---

11. Monitoring & Observability

11.1 Metrics to Track

Event Store:

- Events per second
- Write latency
- Storage growth rate
- Aggregate sizes

Projections:

- Processing lag (time behind event store)
- Rebuild duration
- Query performance

Business Metrics:

- Transaction volume
- Failed transactions
- Average portfolio value

- User activity

11.2 Event Stream Monitoring

Alerting:

- Projection lag exceeds threshold
- Failed event processing
- Unusual event patterns (fraud detection)
- Bank API failures

---

12. Development Workflow

12.1 Event Versioning

Schema Evolution:

```
GoldPurchaseCompleted v1:
{
  purchaseId: string
  goldAmount: number
}

GoldPurchaseCompleted v2:
{
  purchaseId: string
  goldAmount: number
  certificateNumber: string  // Added field
}
```

Handling:

- Upcasting: Convert old events to new format on read
- Version field in event metadata
- Support multiple versions in code

12.2 Testing Strategy

Event-Based Testing:

- Given: Previous events
- When: Command executed

- Then: Expected events emitted

Example Test:

```
Given:
  - UserRegistered
  - WalletCreated
  - FundsDeposited(amount: 100)

When:
  - PurchaseGold(goldAmount: 10g)

Then:
  - GoldPurchaseInitiated
  - FundsReserved(amount: 80)
  - GoldAllocated(amount: 10g)
  - GoldPurchaseCompleted
```

12.3 Development Environment

Local Setup:

- Docker Compose with PostgreSQL + Redis
- Event store seeded with test events
- Mock bank API
- Sample projections

---

13. Deployment Architecture

13.1 Production Infrastructure

Cloud Provider Recommendations:

- AWS: RDS (PostgreSQL), ElastiCache (Redis), ECS/EKS
- Google Cloud: Cloud SQL, Memorystore, GKE
- Azure: Azure Database for PostgreSQL, Azure Cache for Redis, AKS

13.2 High Availability

Database:

- Primary-replica setup
- Automatic failover

- Cross-region replication for disaster recovery

Services:

- Multi-AZ deployment
- Health checks and auto-restart
- Circuit breakers for external dependencies

13.3 Deployment Process

Blue-Green Deployment:

1. Deploy new version alongside current
2. Route small traffic percentage to new version
3. Monitor error rates
4. Gradually shift traffic
5. Rollback capability at any point

---

14. Regulatory Considerations

14.1 Financial Regulations

Requirements:

- Transaction audit trail (Event sourcing provides this)
- Customer identification (KYC)
- Anti-money laundering (AML) monitoring
- Regular financial reporting
- Data retention policies

14.2 Event Retention

Strategy:

- Hot storage: Recent 2 years (fast access)
- Warm storage: 3-7 years (slower access)
- Cold archive: 7+ years (compliance only)

---

15. Next Steps for Implementation

Phase 1: Foundation (Weeks 1-4)

- Set up PostgreSQL event store
- Implement core events
- Build User and Wallet aggregates
- Create API Gateway with authentication

Phase 2: Core Features (Weeks 5-8)

- Bank integration (mock initially)
- Gold purchase flow
- Basic dashboard projections
- Transaction history

Phase 3: Market Integration (Weeks 9-10)

- Real-time price feeds
- Portfolio calculator
- Returns estimation

Phase 4: Advanced Features (Weeks 11-12)

- Pooled investments
- Withdrawals
- Notifications
- Admin panel

Phase 5: Production Readiness (Weeks 13-16)

- Load testing
- Security audit
- Monitoring setup
- Documentation
- Compliance review

---

16. Key Takeaways

Why Event Sourcing for This Platform:

1. **Audit Trail**: Financial regulations require complete transaction history
2. **Debugging**: Can replay events to understand any issue
3. **Temporal Queries**: "What was user's balance on date X?"
4. **Scalability**: Separate read/write optimization
5. **Business Intelligence**: Rich data for analytics
6. **Trust**: Immutable record builds user confidence

Database Choice:

- **PostgreSQL** for event store and read models
- **Redis** for caching and real-time features
- **Kafka** (optional) for event distribution at scale

Critical Success Factors:

- Proper event design (immutable, descriptive)
- Robust idempotency handling
- Comprehensive monitoring
- Regular projection rebuilds testing
- Clear event versioning strategy

Appendix: Technology Stack Summary

Event Store & Read Models:

- PostgreSQL 15+ with JSONB support
- Connection pooling (PgBouncer)

Caching:

- Redis 7+ (cluster mode for production)

API Layer:

- API Gateway: Node.js/Go/Python with JWT
- RESTful APIs + WebSocket for real-time

Event Processing:

- Message Queue: Kafka or RabbitMQ
- Event handlers: Microservices in containerized environment

Infrastructure:

- Container orchestration: Kubernetes or ECS
- Load balancing: Application Load Balancer
- Monitoring: Prometheus + Grafana
- Logging: ELK Stack or cloud-native solutions
- Secrets management: AWS Secrets Manager / HashiCorp Vault

External Integrations:

- Bank API: REST with webhook callbacks
- Gold price data: Third-party market data APIs
- SMS/Email: Twilio, SendGrid