

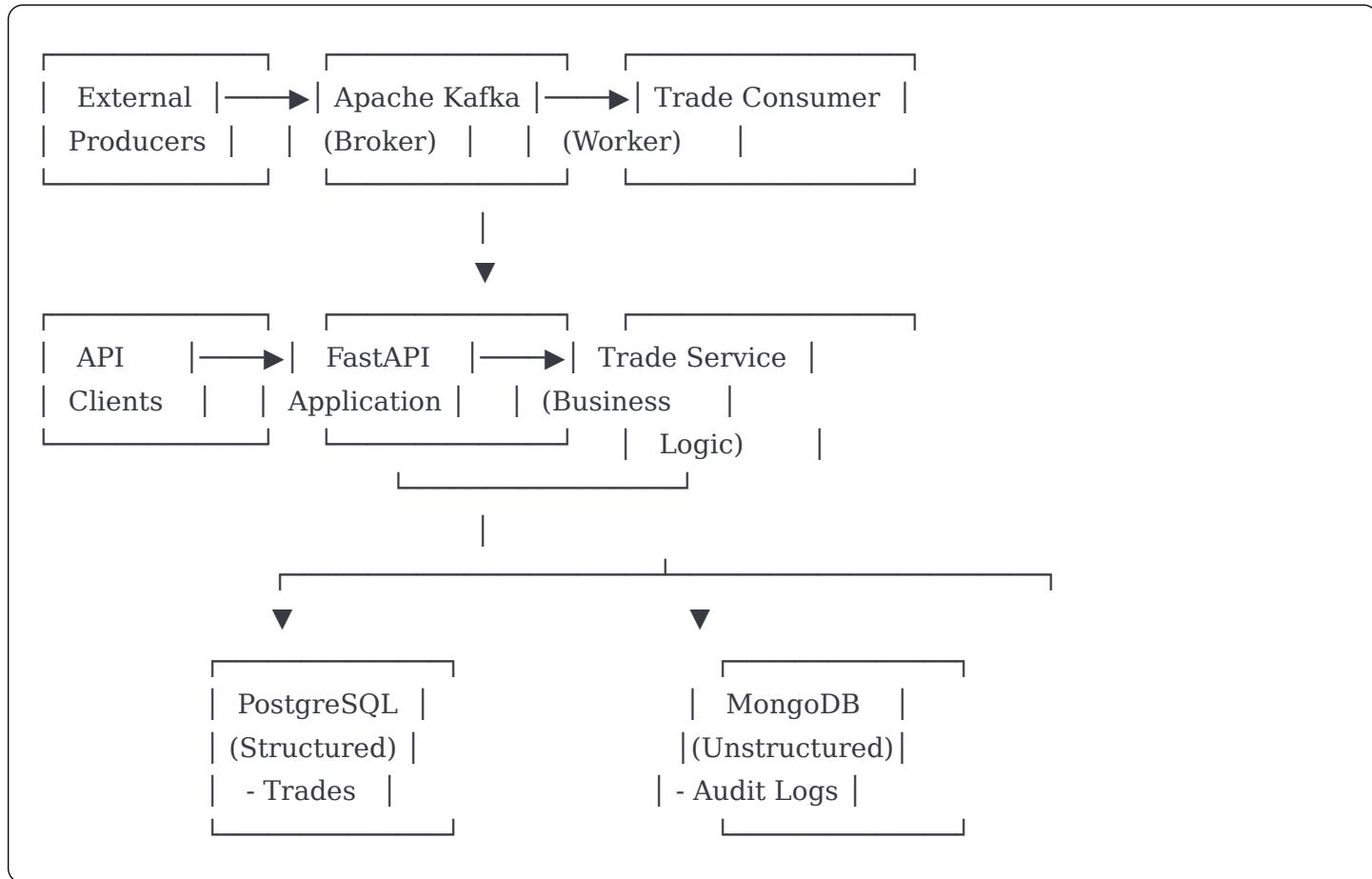
Trade Store - System Architecture

Comprehensive architecture documentation for the Trade Store application.

Architecture Overview

Trade Store is a distributed system designed for high-throughput trade ingestion, validation, and storage. It follows a microservices architecture with event-driven patterns.

High-Level Architecture



System Components

1. API Layer (FastAPI)

Responsibility: HTTP REST API for trade management

Key Files:

- `app/main.py`: Application entry point
- `app/api/routes.py`: API endpoints
- `app/models/schemas.py`: Request/response models

Features:

- OpenAPI/Swagger documentation
- Automatic validation with Pydantic
- CORS support
- Health check endpoints
- Error handling middleware

Endpoints:

```
POST /api/v1/trades      - Create/update trade
GET  /api/v1/trades      - List trades (paginated)
GET  /api/v1/trades/{id}  - Get specific trade
DELETE /api/v1/trades/{id} - Delete trade
GET  /api/v1/trades/book/{id}- Get trades by book
GET  /api/v1/health       - Health check
GET  /api/v1/statistics   - System statistics
POST /api/v1/expire-trades - Manual expiry trigger
GET  /api/v1/audit-logs    - Audit trail
```

2. Business Logic Layer (Services)

Responsibility: Core business rules and validation

Components:

TradeService ([app/services/trade_service.py](#))

- Implements all validation rules
- Orchestrates data persistence
- Manages audit logging
- Provides statistics

Validation Rules:

python

1. Version Control:

- Lower version → Reject (InvalidVersionException)
- Same version → Replace existing
- Higher version → Accept

2. Maturity Date:

- Past date → Reject (InvalidMaturityDateException)
- Today or future → Accept

3. Automatic Expiry:

- maturity_date < current_date → Set expired = true

KafkaTradeConsumer ([app/services/kafka_consumer.py](#))

- Consumes messages from Kafka
- Parses JSON trade data
- Processes through TradeService
- Handles errors and retries
- Commits offsets

ExpiryScheduler ([app/services/expiry_scheduler.py](#))

- Background job scheduler
- Periodic expiry checks
- Configurable interval
- Event logging

3. Data Access Layer (Repositories)

Responsibility: Database abstraction and CRUD operations

PostgresRepository ([app/repositories/postgres_repository.py](#))

Schema:

sql

```
CREATE TABLE trades (
    trade_id VARCHAR(50) PRIMARY KEY,
    version INTEGER NOT NULL,
    counter_party_id VARCHAR(50) NOT NULL,
    book_id VARCHAR(50) NOT NULL,
    maturity_date DATE NOT NULL,
    created_date DATE NOT NULL,
    expired BOOLEAN DEFAULT FALSE,
    last_updated TIMESTAMP
);

-- Indexes for performance
CREATE INDEX idx_trades_book_id ON trades(book_id);
CREATE INDEX idx_trades_maturity_date ON trades(maturity_date);
CREATE INDEX idx_trades_expired ON trades(expired);
CREATE INDEX idx_trades_counter_party ON trades(counter_party_id);
```

Operations:

- CRUD for trades
- Bulk expiry updates
- Query by book, status
- Connection pooling
- Transaction management

MongoDBRepository ([app/repositories/mongodb_repository.py](#))

Collections:

```

javascript

// audit_logs
{
  _id: ObjectId,
  trade_id: String,
  action: String, // CREATE, UPDATE, DELETE, VALIDATION_FAILED
  status: String, // success, failed
  details: Object,
  timestamp: DateTime
}

// trade_events
{
  _id: ObjectId,
  event_type: String, // TRADES_EXPIRED, KAFKA_ERROR, etc.
  severity: String, // info, warning, error
  data: Object,
  timestamp: DateTime
}

```

Operations:

- Audit log persistence
- Event logging
- Log retrieval with filtering
- Log cleanup (retention policy)

4. Message Broker (Apache Kafka)

Configuration:

```

yaml

Topic: trades
Partitions: 3
Replication Factor: 1 (dev), 3 (prod)
Retention: 7 days

```

Message Format:

```
json
```

```
{  
  "trade_id": "T1",  
  "version": 1,  
  "counter_party_id": "CP-1",  
  "book_id": "B1",  
  "maturity_date": "2025-05-20",  
  "created_date": "2024-12-11",  
  "expired": false  
}
```

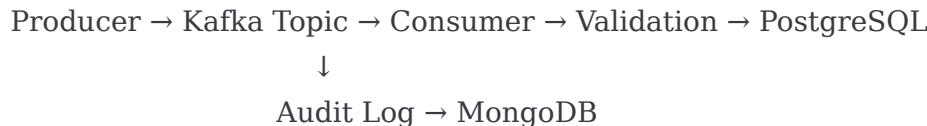
Consumer Configuration:

```
python
```

```
{  
  'group.id': 'trade-consumer-group',  
  'auto.offset.reset': 'earliest',  
  'enable.auto.commit': True,  
  'session.timeout.ms': 6000  
}
```

>Data Flow Patterns

Pattern 1: Trade Creation via Kafka



Sequence:

1. External system publishes trade to Kafka
2. Consumer polls and receives message
3. Consumer parses JSON to TradeCreate object
4. TradeService validates against business rules
5. If valid: Save to PostgreSQL, log to MongoDB
6. If invalid: Log error to MongoDB, skip commit
7. Consumer commits offset

Pattern 2: Trade Creation via REST API

Client → FastAPI → TradeService → Validation → PostgreSQL



Audit Log → MongoDB

Sequence:

1. Client sends POST request with trade data
2. FastAPI validates request schema (Pydantic)
3. TradeService validates business rules
4. If valid: Save to PostgreSQL, log to MongoDB, return 201
5. If invalid: Return 400 with error details

Pattern 3: Scheduled Expiry Check

Scheduler (Cron) → TradeService → PostgreSQL (Bulk Update)



Event Log → MongoDB

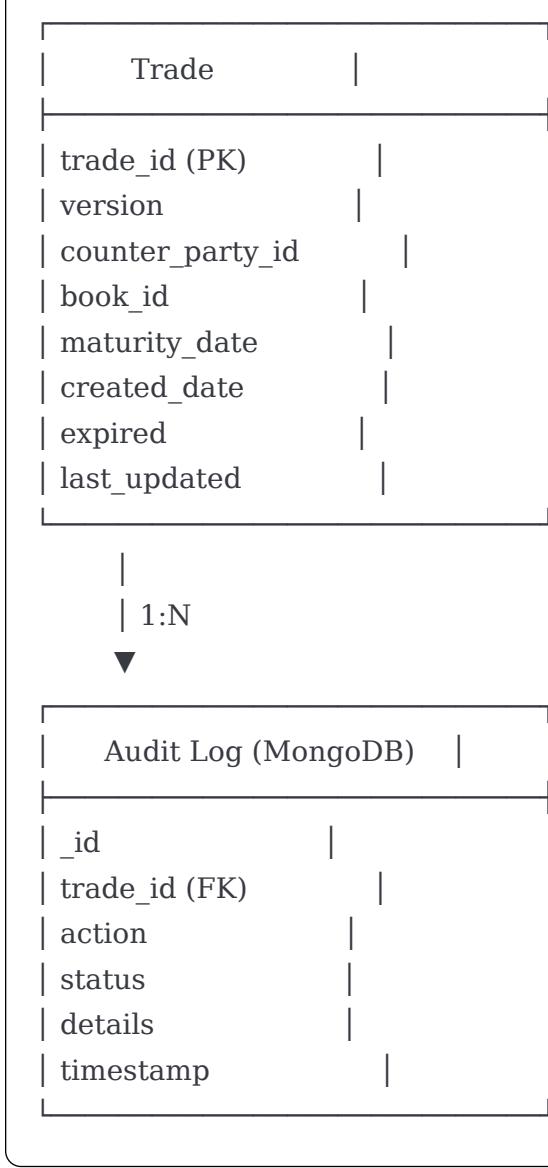
Sequence:

1. APScheduler triggers at configured interval
2. TradeService queries trades with `maturity_date < today`
3. Bulk update expired flag in PostgreSQL
4. Log expiry event to MongoDB
5. Return count of updated trades



Database Design

Entity-Relationship Diagram



Data Consistency Strategy

PostgreSQL (ACID):

- Transactions for data integrity
- Foreign key constraints
- Unique constraints on trade_id
- Isolation level: Read Committed

MongoDB (BASE):

- Eventually consistent
- No foreign keys (denormalized)
- Indexed on trade_id and timestamp
- Append-only (no updates/deletes)

Cross-Database Consistency:

- PostgreSQL is source of truth
- MongoDB for audit/analytics only
- No distributed transactions
- Compensating transactions on failure

🔒 Security Architecture

Authentication & Authorization

- Environment-based configuration
- No hardcoded credentials
- Secrets management (future: Vault/AWS Secrets)

Data Protection

- PostgreSQL: SSL connections (production)
- MongoDB: Authentication enabled (production)
- Kafka: SASL/SSL (production)
- API: HTTPS only (production)

Input Validation

- Pydantic schemas for API
- Type checking throughout
- SQL injection prevention (ORM)
- NoSQL injection prevention (parameterized)

Audit Trail

- All operations logged
- Immutable audit logs
- Timestamp on all records
- Failed attempts logged

⚡ Performance Optimization

Database Optimization

```
sql

-- Indexes for common queries
CREATE INDEX idx_trades_composite
ON trades(book_id, expired, maturity_date);

-- Analyze query performance
EXPLAIN ANALYZE
SELECT * FROM trades
WHERE book_id = 'B1' AND expired = false;
```

Connection Pooling

```
python

# PostgreSQL connection pool
pool_size = 10
max_overflow = 20
pool_pre_ping = True # Verify connections

# Reuse connections efficiently
```

Caching Strategy

```
Level 1: Application memory (for reference data)
Level 2: Redis (future enhancement)
Level 3: PostgreSQL query cache
```

Kafka Optimization

```
python

# Consumer configuration
{
    'fetch.min.bytes': 1024,
    'fetch.wait.max.ms': 500,
    'max.partition.fetch.bytes': 1048576
}

# Batch processing
messages = consumer.consume(num_messages=100)
```

Scalability Design

Horizontal Scaling

API Layer:

```
yaml  
  
# Load balanced across multiple instances  
services:  
  trade-app:  
    deploy:  
      replicas: 3  
    ports:  
      - "8000-8002:8000"
```

Consumer Layer:

```
yaml  
  
# Multiple consumers in same group  
# Kafka automatically distributes partitions  
services:  
  trade-consumer:  
    deploy:  
      replicas: 3
```

Database Layer:

```
PostgreSQL: Read replicas for queries  
MongoDB: Sharding by trade_id (if needed)
```

Vertical Scaling

```
yaml  
  
# Increase resources per container  
services:  
  trade-app:  
    cpus: '2.0'  
    mem_limit: 4g
```

Auto-Scaling (Kubernetes)

```
yaml
```

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: trade-app
spec:
  minReplicas: 3
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

Disaster Recovery

Backup Strategy

PostgreSQL:

```
bash

# Daily full backups
pg_dump -U postgres tradestore > backup_$(date +%Y%m%d).sql

# Continuous WAL archiving (production)
archive_mode = on
archive_command = 'cp %p /backups/wal/%f'
```

MongoDB:

```
bash

# Daily backups
mongodump --db tradestore --out /backups/mongo_$(date +%Y%m%d)

# Oplog backup for point-in-time recovery
```

Recovery Procedures

Data Loss Scenarios:

1. **Recent data loss (<24h):** Restore from previous backup + replay Kafka messages
2. **Database corruption:** Restore from backup, reindex
3. **Complete failure:** Restore databases, restart services

RTO/RPO Targets:

- RTO (Recovery Time Objective): 1 hour
- RPO (Recovery Point Objective): 5 minutes (with Kafka replay)

Monitoring & Observability

Metrics to Monitor

Application Metrics:

- Request rate (requests/sec)
- Response time (p50, p95, p99)
- Error rate (4xx, 5xx)
- Active connections

Business Metrics:

- Trades processed/sec
- Validation failure rate
- Expired trades count
- Average processing time

Infrastructure Metrics:

- CPU usage
- Memory usage
- Disk I/O
- Network throughput

Logging Strategy

Log Levels:

```
python
```

DEBUG: Development only

INFO: Normal operations (trade created, etc.)

WARNING: Validation failures, retries

ERROR: Service failures, exceptions

CRITICAL: System-wide failures

Log Aggregation:

Application → Structured logs → ELK Stack / CloudWatch

→ Alerting rules

→ Dashboards

Health Checks

```
python
```

Application health

```
GET /api/v1/health
```

```
{
```

```
  "status": "healthy",
```

```
  "services": {
```

```
    "postgres": "up",
```

```
    "mongodb": "up",
```

```
    "kafka": "up"
```

```
}
```

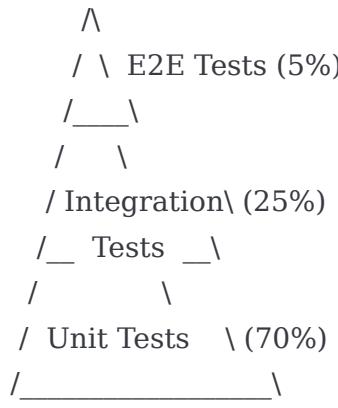
```
}
```

Liveness probe: Is service running?

Readiness probe: Can service accept traffic?

Testing Strategy

Test Pyramid



Test Categories

Unit Tests:

- Service layer logic
- Validation rules
- Repository methods
- Utilities

Integration Tests:

- API endpoints
- Database operations
- Kafka integration
- Service interactions

Performance Tests:

- Load testing
- Stress testing
- Spike testing
- Endurance testing

🚀 Deployment Architecture

Development Environment

docker-compose.yml → Local containers → Hot reload enabled

Production Environment

```
Kubernetes Cluster
├─ Namespace: trade-store-prod
├─ Deployments:
|   ├─ trade-app (3 replicas)
|   ├─ trade-consumer (3 replicas)
|   └─ expiry-scheduler (1 replica)
└─ Services:
    ├─ trade-app-svc (LoadBalancer)
    └─ Internal services (ClusterIP)
└─ ConfigMaps: Environment config
└─ Secrets: Credentials
└─ PersistentVolumes: Database storage
```

CI/CD Pipeline



Technology Decisions

Why FastAPI?

- High performance (async/await)
- Automatic OpenAPI docs
- Built-in validation
- Type hints support
- Active community

Why Kafka?

- High throughput
- Fault tolerant
- Horizontal scalability
- Message replay capability
- Industry standard

Why PostgreSQL + MongoDB?

- **PostgreSQL:** ACID compliance for trade data
- **MongoDB:** Flexible schema for audit logs
- Different access patterns
- Optimal for each use case

Why Docker?

- Consistent environments
- Easy deployment
- Isolation
- Portability
- Resource efficiency

Future Enhancements

1. **Message Deduplication:** Prevent duplicate trade processing
2. **Distributed Tracing:** OpenTelemetry integration
3. **GraphQL API:** Alternative to REST
4. **Event Sourcing:** Full audit trail of all changes
5. **CQRS Pattern:** Separate read/write models
6. **Multi-region:** Geographic distribution
7. **ML Integration:** Anomaly detection
8. **Real-time Analytics:** Stream processing

Last Updated: December 2024 **Version:** 1.0.0