projects current info

📖 Detailed Project Descriptions (For Resume/Interview)

🚀 Project 1: Transaction Processing Middleware - Complete Description

Project Overview:

Designed and developed a critical middleware service that acts as a secure intermediary between multiple banking channels (ATM, Mobile Banking, Internet Banking) and the core banking system. The system handles thousands of concurrent transaction requests, performing real-time validation, enrichment, fraud checks, and routing while maintaining strict security, audit compliance, and high availability.

Business Problem Solved:

Previously, each banking channel had direct integration with the core system, creating:

- **Tight coupling** making changes risky and slow
- **Inconsistent validation** across different channels
- **Duplicate code** for common operations
- **Difficult monitoring** and troubleshooting
- **Security vulnerabilities** from multiple access points

Solution Delivered:

Built a centralized middleware that:

- **Standardizes** all transaction processing logic
- **Enforces** consistent security and validation rules
- **Provides** unified monitoring and logging
- **Enables** easy addition of new banking channels
- **Reduces** core banking system load by 40%

Technical Architecture:

Layer 1 - API Layer (What I Built):

- RESTful endpoints using **JAX-RS (Jersey framework)**
- HTTP Servlets for request/response handling
- JWT token validation for authentication
- Input validation and sanitization
- Rate limiting to prevent abuse

- CORS configuration for web clients

Layer 2 - Service Layer (What I Built):

- Business rules engine for transaction validation
- Account balance verification
- Daily/monthly limit checks
- Fraud detection integration
- Transaction enrichment (adding metadata)
- Orchestration of multiple downstream calls
- Exception handling and retry logic

Layer 3 - DAO Layer (What I Built):

- JDBC-based database operations
- Prepared statements for SQL injection prevention
- Connection pooling with HikariCP (optimized for 50-200 concurrent connections)
- Transaction management (ACID compliance)
- Audit logging to separate tables

Database Design:

```
-- Main transaction table
CREATE TABLE transactions (
    transaction_id VARCHAR(50) PRIMARY KEY,
    from_account VARCHAR(20) NOT NULL,
    to_account VARCHAR(20) NOT NULL,
    amount DECIMAL(15,2) NOT NULL,
    transaction_type VARCHAR(20) NOT NULL,
    status VARCHAR(20) NOT NULL,
    channel VARCHAR(20) NOT NULL,
    timestamp TIMESTAMP NOT NULL,
    correlation_id VARCHAR(50) NOT NULL,
    error_message TEXT,
    processing_time_ms INT,
    INDEX idx_from_account (from_account),
    INDEX idx_timestamp_status (timestamp, status),
    INDEX idx_correlation_id (correlation_id)
);

-- Audit log table
CREATE TABLE transaction_audit (
    audit_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    transaction_id VARCHAR(50) NOT NULL,
    action VARCHAR(50) NOT NULL,
    old_status VARCHAR(20),
    new_status VARCHAR(20),
```

```
    modified_by VARCHAR(50),
    modified_at TIMESTAMP NOT NULL,
    details TEXT
);
```

Key Features I Implemented:

1. **Multi-threaded Request Processing:**

```
// Configured thread pool for high throughput
ExecutorService executorService = new ThreadPoolExecutor(
    20,  // core pool size
    100, // maximum pool size
    60L, TimeUnit.SECONDS,
    new LinkedBlockingQueue<>(500),
    new ThreadPoolExecutor.CallerRunsPolicy()
);
```

1. **Intelligent Retry Mechanism:**

- Transient failures: 3 retries with exponential backoff
- Permanent failures: Immediate failure response
- Circuit breaker pattern for downstream service failures

1. **Comprehensive Logging:**

```
// Correlation ID for end-to-end tracing
MDC.put("correlationId", correlationId);
log.info("Transaction initiated: from={}, to={}, amount={}",
        fromAccount, toAccount, amount);
```

1. **Security Implementation:**

- JWT-based authentication with RSA signature verification
- Field-level encryption for sensitive data
- Rate limiting: 100 requests per minute per user
- SQL injection prevention using prepared statements
- XSS protection on all input parameters

Performance Achievements:

- **Throughput:** Handles 500+ transactions per second
- **Latency:** Average response time 180ms (95th percentile: 350ms)
- **Availability:** 99.95% uptime
- **Error Rate:** Reduced from 2.5% to 0.3%

Production Issues I Resolved:

Issue 1: Database Connection Leaks

- **Problem:** Application crashed after 6-8 hours with "Too many connections"
- **Root Cause:** try-catch blocks not properly closing connections on exceptions
- **Solution:** Implemented try-with-resources pattern consistently
- **Result:** Zero connection leaks in production

Issue 2: Thread Deadlock

- **Problem:** Application hung during peak hours
- **Root Cause:** Two services acquiring locks in different order
- **Solution:** Standardized lock acquisition order, added timeout
- **Result:** No deadlocks in 6+ months

Issue 3: Slow Transaction Queries

- **Problem:** Dashboard showed transactions loading slowly
- **Root Cause:** Missing composite index on (from_account, timestamp)
- **Solution:** Added proper indexes and rewrote N+1 queries
- **Result:** Query time reduced from 5s to 120ms

Testing Strategy:

- **Unit Tests:** 85% code coverage using JUnit and Mockito
- **Integration Tests:** End-to-end API testing with embedded database
- **Load Tests:** JMeter scripts simulating 1000 concurrent users
- **Security Tests:** OWASP vulnerability scanning in CI/CD

Deployment & DevOps:

- AWS EC2 instances (t3.large) with auto-scaling
- Application deployed as executable JAR
- Blue-green deployment for zero-downtime releases
- CloudWatch for monitoring and alerting
- ELK stack for centralized logging

Technologies & Tools:

- **Language:** Core Java 11
- **Frameworks:** JAX-RS (Jersey), Servlets 4.0
- **Database:** MySQL 8.0 on AWS RDS
- **Connection Pool:** HikariCP
- **Logging:** SLF4J + Logback
- **Testing:** JUnit 5, Mockito, Postman
- **Build:** Maven

- **Cloud:** AWS (EC2, RDS, CloudWatch)
- **Version Control:** Git, GitHub

Impact Statement for Interview:

"I developed a mission-critical middleware that processes over 1 million transactions daily for Bank of America. By implementing proper architecture, caching, and database optimization, I improved system performance by 60% and reduced error rates by 90%. This system directly supports ATM, mobile, and internet banking operations for millions of customers."

🎯 Project 2: Real-Time Monitoring Dashboard Service - Complete Description

Project Overview:

Developed a high-performance backend service that aggregates, processes, and serves real-time banking metrics and analytics to operations dashboards. The system provides instant visibility into transaction health, system performance, error patterns, and business KPIs, enabling proactive issue detection and rapid response.

Business Problem Solved:

Operations team previously faced:

- **Delayed visibility:** Reports generated overnight, issues discovered too late
- **No real-time monitoring:** Couldn't detect ongoing issues
- **Manual data gathering:** Teams spent hours creating reports
- **Inconsistent metrics:** Different systems showing different numbers
- **Poor incident response:** Took hours to identify root causes

Solution Delivered:

Built a real-time analytics platform that:

- **Provides** live metrics updated every 5 minutes
- **Aggregates** data from multiple source systems
- **Pre-calculates** complex metrics to reduce query load
- **Caches** frequently accessed data for instant response
- **Alerts** operations team on anomalies automatically
- **Reduced** incident detection time from hours to minutes

Technical Architecture:

Component 1 - REST API Layer:

```
// Sample endpoint structure
@Path("/api/metrics")
public class MetricsResource {
```

```
    @GET
    @Path("/hourly-summary")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getHourlySummary(
        @QueryParam("startTime") String start,
        @QueryParam("endTime") String end,
        @QueryParam("channel") String channel) {

        // Validate and parse parameters
        // Check cache
        // Query database if cache miss
        // Return JSON response
    }
}
```

Component 2 - Caching Strategy:

```
public class MetricsCacheManager {
    // Two-level cache: Hot data + warm data
    private ConcurrentHashMap<String, CachedMetric> hotCache;  // 5 min TTL
    private ConcurrentHashMap<String, CachedMetric> warmCache; // 1 hour TTL

    // Cache warming on startup
    public void warmupCache() {
        // Pre-load today's metrics
        // Pre-load most accessed data
        // Background refresh every 5 minutes
    }
}
```

Component 3 - Scheduled Jobs:

```
// Hourly metrics pre-calculation
public class HourlyMetricsJob extends TimerTask {
    @Override
    public void run() {
        // Calculate last hour's metrics
        // Store in metrics_cache table
        // Update aggregated_metrics table
        // Trigger cache refresh
    }
}

// Daily cleanup job
public class DataRetentionJob extends TimerTask {
    @Override
    public void run() {
        // Archive data older than 90 days
```

```
        // Delete raw logs older than 30 days
        // Vacuum/optimize tables
    }
}
```

Database Design:

```
-- Real-time transactions (kept for 30 days)
CREATE TABLE transactions (
    transaction_id VARCHAR(50) PRIMARY KEY,
    amount DECIMAL(15,2),
    status VARCHAR(20),
    channel VARCHAR(20),
    timestamp TIMESTAMP,
    processing_time_ms INT,
    error_code VARCHAR(10),
    INDEX idx_timestamp (timestamp),
    INDEX idx_status_timestamp (status, timestamp),
    INDEX idx_channel_timestamp (channel, timestamp)
) PARTITION BY RANGE (TO_DAYS(timestamp)) (
    PARTITION p_old VALUES LESS THAN (TO_DAYS('2025-12-01')),
    PARTITION p_current VALUES LESS THAN MAXVALUE
);

-- Pre-calculated hourly metrics (kept for 1 year)
CREATE TABLE hourly_metrics (
    metric_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    hour_start TIMESTAMP NOT NULL,
    channel VARCHAR(20) NOT NULL,
    total_transactions INT NOT NULL,
    successful_transactions INT NOT NULL,
    failed_transactions INT NOT NULL,
    avg_processing_time_ms INT NOT NULL,
    p95_processing_time_ms INT NOT NULL,
    total_amount DECIMAL(15,2) NOT NULL,
    unique_users INT NOT NULL,
    calculated_at TIMESTAMP NOT NULL,
    UNIQUE KEY uk_hour_channel (hour_start, channel),
    INDEX idx_hour_start (hour_start)
);

-- Cached metrics for instant retrieval
CREATE TABLE metrics_cache (
    cache_key VARCHAR(100) PRIMARY KEY,
    metric_data JSON NOT NULL,
    created_at TIMESTAMP NOT NULL,
    expires_at TIMESTAMP NOT NULL,
```

```
    INDEX idx_expires (expires_at)
);
```

Key Features I Implemented:

1. **Smart Query Optimization:**

```
-- Before: 8-10 seconds (full table scan)
SELECT COUNT(*), AVG(amount)
FROM transactions
WHERE timestamp BETWEEN ? AND ?;

-- After: 200ms (uses index + pre-calculated data)
SELECT SUM(total_transactions), SUM(total_amount)/SUM(total_transactions)
FROM hourly_metrics
WHERE hour_start BETWEEN ? AND ?;
```

1. **Intelligent Caching:**

- **Hot data:** Current day metrics (5 min TTL)
- **Warm data:** Last 7 days (1 hour TTL)
- **Cold data:** Query database (cache result)
- **Cache hit ratio:** 92%

1. **Concurrent Request Handling:**

```
// Handle multiple dashboard users simultaneously
public class MetricsService {
    private final ExecutorService executor = Executors.newFixedThreadPool(50);

    public CompletableFuture<MetricsSummary> getMetricsAsync(String key) {
        return CompletableFuture.supplyAsync(() ->
            getMetricsInternal(key), executor);
    }
}
```

1. **Anomaly Detection:**

```
// Alert on significant deviations
if(currentHourFailureRate > historicalAvg * 1.5) {
    alertService.sendAlert(
        "High failure rate detected: " + currentHourFailureRate + "%",
        Severity.HIGH
    );
}
```

Performance Optimizations I Implemented:

Optimization 1: Query Indexing

```
-- Added composite indexes
CREATE INDEX idx_composite_1 ON transactions(timestamp, status, channel);
CREATE INDEX idx_composite_2 ON transactions(channel, timestamp)
   INCLUDE (amount, processing_time_ms);

-- Result: Query time 8s → 180ms (97% improvement)
```

Optimization 2: Connection Pooling

```
HikariConfig config = new HikariConfig();
config.setMaximumPoolSize(50);          // Match thread pool size
config.setMinimumIdle(10);              // Keep connections ready
config.setConnectionTimeout(2000);      // Fail fast
config.setIdleTimeout(600000);          // 10 minutes
config.setMaxLifetime(1800000);         // 30 minutes
config.setLeakDetectionThreshold(60000); // Detect leaks

// Result: Eliminated "connection timeout" errors
```

Optimization 3: Batch Processing

```
// Instead of 1000 individual inserts
// Use batch insert for pre-calculated metrics
String sql = "INSERT INTO hourly_metrics (hour_start, channel, ...) VALUES (?, ?, ...)";
try (PreparedStatement stmt = conn.prepareStatement(sql)) {
   for(MetricData metric : metrics) {
      stmt.setTimestamp(1, metric.getHourStart());
      stmt.setString(2, metric.getChannel());
      // ... set other parameters
      stmt.addBatch();
   }
   stmt.executeBatch();
}

// Result: 1000 inserts in 2 seconds instead of 45 seconds
```

Production Issues I Resolved:

Issue 1: OutOfMemoryError

- **Problem:** Application crashed every 24-48 hours
- **Investigation:** Captured heap dump, analyzed with Eclipse MAT
- **Root Cause:** ConcurrentHashMap cache growing unbounded (500K+ entries)
- **Solution:** Implemented TTL-based cleanup thread + size limits
- **Result:** Memory stable at 2GB, zero OOM errors in 4 months

Issue 2: Dashboard Loading Slowly

- **Problem:** Dashboard took 8-12 seconds to load
- **Investigation:** Enabled query logging, ran EXPLAIN on slow queries
- **Root Cause:** Missing indexes + N+1 query problem + no caching
- **Solution:** Added indexes, rewrote queries, implemented cache layer
- **Result:** Load time reduced to 300ms (96% improvement)

Issue 3: Stale Cache Data

- **Problem:** Dashboard showing old data after transactions
- **Root Cause:** Cache not invalidated when new data arrived
- **Solution:** Implemented event-based cache invalidation
- **Result:** Data freshness within 5 seconds

Monitoring & Observability:

1. **Application Metrics:**
   - Request rate, error rate, latency (p50, p95, p99)
   - Cache hit/miss ratio
   - Database connection pool usage
   - Thread pool queue size
2. **Business Metrics:**
   - Transactions per second by channel
   - Success/failure rates
   - Average transaction amount
   - Peak load times
3. **Alerts Configured:**
   - Error rate > 1% (warning)
   - Error rate > 5% (critical)
   - Response time > 1s (warning)
   - Cache hit ratio < 80% (info)
   - Database connections > 80% (warning)

Testing Strategy:

- **Unit Tests:** 80% coverage for service and DAO layers
- **Integration Tests:** API endpoint testing with test database
- **Performance Tests:** Load tested with 500 concurrent users
- **Stress Tests:** Verified graceful degradation under extreme load

Technologies & Tools:

- **Language:** Core Java 11
- **Framework:** JAX-RS (Jersey)
- **Database:** MySQL 8.0 (with partitioning)
- **Caching:** ConcurrentHashMap, Caffeine Cache
- **Scheduler:** Java TimerTask, ScheduledExecutorService
- **Connection Pool:** HikariCP
- **Testing:** JUnit 5, Mockito, JMeter
- **Monitoring:** CloudWatch, custom JMX metrics
- **Build:** Maven
- **Deployment:** AWS EC2, RDS

Impact Statement for Interview:

"I built a real-time analytics platform that provides instant visibility into banking operations, processing metrics for over 1 million daily transactions. Through intelligent caching, query optimization, and scheduled pre-calculations, I reduced dashboard load times by 96% and enabled the operations team to detect and resolve issues 10x faster. The system handles 500+ concurrent users with 99.9% availability."

🎯 Combined Impact Summary

Both projects together demonstrate my ability to:

✅ **Design & Build:** Scalable Java backend systems from scratch

✅ **Optimize:** Database queries and application performance

✅ **Debug:** Complex production issues using systematic approaches

✅ **Secure:** Banking applications with industry-standard practices

✅ **Test:** Comprehensive testing strategies for reliability

✅ **Deploy:** Cloud-based solutions with proper DevOps practices

✅ **Collaborate:** Work with cross-functional teams effectively

✅ **Deliver:** Measurable business value and system improvements

Key Achievements:

- 🚀 **1M+ transactions** processed daily
- ⚡ **96% performance improvement** in dashboard load times
- 🔒 **99.95% uptime** maintained across services
- 🐞 **90% reduction** in production error rates
- ⏱️ **60% faster** transaction processing
- 💾 **Zero memory leaks** after optimization
- 📊 **Real-time monitoring** enabling proactive issue resolution