Notification Queue Architecture - Queue-First Design

Overview

Service 2 uses a **queue-first architecture** for all notification operations, routing requests through a persistent Redis queue before processing. This provides complete crash recovery protection for both Service 1 and Service 2 failures.

Architecture Diagram

```
┌─────────────────────────────────────────────────────────────────┐
│                    SERVICE 2 (Business Logic)                   │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  API Endpoints:                                                 │
│  • POST /send-to-user          (authenticated)                 │
│  • POST /send-to-users                                         │
│  • POST /send-bulk                                             │
│                                                                 │
│         ↓ ALL notifications go here first                      │
│                                                                 │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │            REDIS QUEUE (Persistent)                       │ │
│  │  • FIFO processing                                        │ │
│  │  • Survives Service 2 restart                             │ │
│  │  • Automatic retry (3 attempts)                           │ │
│  │  • Configurable via ENABLE_NOTIFICATION_QUEUE             │ │
│  └───────────────────────────────────────────────────────────┘  │
│                                                                 │
│         ↓ Queue processor dequeues one-by-one                  │
│                                                                 │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │     Notification Client (HTTP with retry)                 │ │
│  │     • Makes HTTP POST to Service 1                        │
│  │     • Internal API key authentication                     │ │
│  │     • Auto-retry on failure                               │ │
│  └───────────────────────────────────────────────────────────┘  │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
                              │
                              │ HTTP POST
                              │ x-internal-api-key: <KEY>
                              ↓
┌─────────────────────────────────────────────────────────────────┐
│                    SERVICE 1 (Firebase Handler)                 │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  • Receives notification requests                              │
│  • Fetches user FCM tokens from MongoDB                        │
│  • Sends via Firebase Cloud Messaging                  │
│  • Returns success/failure response                    │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

How It Works

Queue-First Approach (Default)

When `ENABLE_NOTIFICATION_QUEUE=true` (default):

1. **API Request Received** → Service 2 endpoint called
2. **Immediate Queueing** → Notification added to Redis queue
3. **Success Response** → API returns `{success: true, queued: true}`
4. **Background Processing** → Queue processor dequeues and sends to Service 1
5. **Automatic Retry** → If Service 1 fails, retry up to 3 times

Direct HTTP Mode (Fallback)

When `ENABLE_NOTIFICATION_QUEUE=false`:

1. **API Request Received** → Service 2 endpoint called
2. **Direct HTTP Call** → Send immediately to Service 1
3. **Synchronous Response** → Wait for Service 1 response
4. **No Retry** → Failure = lost notification

Crash Recovery Scenarios

Scenario 1: Service 2 Crash (Business Logic)

Problem: Service 2 crashes while processing notifications

Protection:
☑ **Queue survives** - All queued notifications persist in Redis
☑ **No data loss** - Notifications waiting in queue are safe
☑ **Auto-resume** - Queue processor restarts when service restarts

Example:

```
# Before crash: 100 notifications queued
Service 2 crashes at 11:30 AM
Redis keeps queue intact

# After restart at 11:32 AM
Service 2 starts → Queue processor resumes
Processes remaining 100 notifications automatically
```

Scenario 2: Service 1 Crash (Firebase Handler)

Problem: Service 1 crashes and can't receive notifications

Protection:
☑ **Queue retains notifications** - Service 2 queue keeps trying
☑ **Automatic retry** - 3 retry attempts with exponential backoff
☑ **Failed queue** - After max retries, moved to failed queue for manual review

Example:

```
# Service 1 crashes at 2:00 PM
Queue processor attempts to send → fails
Retry 1: Waits 5 seconds → fails
Retry 2: Waits 10 seconds → fails
Retry 3: Waits 20 seconds → fails
Notification moved to failed queue

# Service 1 restarts at 2:05 PM
Queue processor continues with new notifications
Manual review of failed queue required for old notifications
```

Scenario 3: Both Services Crash

Problem: Both Service 1 and Service 2 crash simultaneously

Protection:
☑ **Redis persistence** - Queue survives (if Redis has persistence enabled)
☑ **Complete recovery** - Both services resume processing after restart

Example:

```
# Both services crash at 3:00 PM
# 500 notifications in Redis queue

# Services restart at 3:10 PM
Service 2 → Queue processor resumes
Service 1 → Ready to receive requests
All 500 notifications processed successfully
```

Configuration

Environment Variables (.env)

```
# Enable queue-first architecture (default: true)
ENABLE_NOTIFICATION_QUEUE=true

# Service 1 URL
SERVICE_1_URL=http://localhost:5000

# Internal service authentication
INTERNAL_SERVICE_API_KEY=your_secret_key_here

# Redis configuration
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_PASSWORD=your_redis_password
```

Redis Persistence (Recommended)

Create `redis.conf` with the following settings:

```
# RDB Persistence (snapshot)
save 900 1      # Save if 1 key changed in 900 seconds
save 300 10     # Save if 10 keys changed in 300 seconds
save 60 10000   # Save if 10000 keys changed in 60 seconds

# AOF Persistence (append-only file - RECOMMENDED)
appendonly yes
appendfilename "notification-queue.aof"
appendfsync everysec  # Sync every second (balance between performance and durability)

# Auto-rewrite AOF to prevent file bloat
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

Start Redis with config:

```
redis-server /path/to/redis.conf
```

Testing Crash Recovery

Test 1: Service 2 Crash Recovery

```
# Terminal 1: Start Service 2
cd PHRMA-PRODUCTION-APP-BACKEND-MAIN-2
npm start
```

```
# Terminal 2: Send notifications
curl -X POST http://localhost:5002/api/v1/notifications/send-to-user \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <JWT_TOKEN>" \
  -d '{
    "title": "Test Notification",
    "body": "Testing crash recovery"
  }'

# Verify queued
# Response: {"success": true, "queued": true}

# CRASH SERVICE 2 (Ctrl+C in Terminal 1)

# Terminal 3: Check Redis queue
redis-cli LLEN notification:queue
# Should show 1 notification

# Restart Service 2 (Terminal 1)
npm start

# Verify: Notification should be processed automatically
# Check Service 1 logs for successful delivery
```

Test 2: Service 1 Crash Recovery

```
# Terminal 1: Start Service 1
cd PHRMA-PRODUCTION-APP-BACKEND-MAIN
npm start

# Terminal 2: Start Service 2
cd PHRMA-PRODUCTION-APP-BACKEND-MAIN-2
npm start

# Terminal 3: Send notification
curl -X POST http://localhost:5002/api/v1/notifications/send-to-user \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <JWT_TOKEN>" \
  -d '{
    "title": "Test Notification",
    "body": "Testing Service 1 crash"
  }'

# CRASH SERVICE 1 immediately (Ctrl+C in Terminal 1)

# Check Service 2 logs - should see retry attempts:
# "📤 Notification request to Service 1: POST /send-to-user"
# "❌ No response from Service 1 notification service"
# "🔄 Retry attempt 1/3..."
# "🔄 Retry attempt 2/3..."
# "🔄 Retry attempt 3/3..."

# Restart Service 1 (Terminal 1)
npm start

# New notifications will now process successfully
# Check failed queue for old notifications:
redis-cli LLEN notification:failed
```

Test 3: Complete System Recovery

```
# 1. Start both services
cd PHRMA-PRODUCTION-APP-BACKEND-MAIN && npm start &
cd PHRMA-PRODUCTION-APP-BACKEND-MAIN-2 && npm start &

# 2. Queue multiple notifications
for i in {1..10}; do
  curl -X POST http://localhost:5002/api/v1/notifications/send-to-user \
    -H "Authorization: Bearer <JWT_TOKEN>" \
    -H "Content-Type: application/json" \
    -d "{\"title\": \"Test $i\", \"body\": \"Message $i\"}"
done

# 3. Check queue status
curl http://localhost:5002/api/v1/notifications/queue-stats

# 4. Crash both services (Ctrl+C both)

# 5. Verify queue persistence
redis-cli LLEN notification:queue
# Should show approximate count

# 6. Restart both services
cd PHRMA-PRODUCTION-APP-BACKEND-MAIN && npm start &
cd PHRMA-PRODUCTION-APP-BACKEND-MAIN-2 && npm start &

# 7. Verify processing
# Check both service logs for successful deliveries
```

Benefits of Queue-First Architecture

Feature	Without Queue	With Queue (Queue-First)
Service 2 Crash	❌ Notifications lost	✅ Persist in Redis

Feature	Without Queue	With Queue (Queue-First)
Service 1 Crash	✗ Notifications lost	☑ Auto-retry 3 times
Network Issues	✗ Fail immediately	☑ Retry with backoff
High Load	✗ Blocks API response	☑ Async processing
Monitoring	✗ No visibility	☑ Queue stats available
Recovery	✗ Manual resend	☑ Automatic

Queue Monitoring

Check Queue Status

```
# Queue length
redis-cli LLEN notification:queue

# Processing queue
redis-cli LLEN notification:processing

# Failed notifications
redis-cli LLEN notification:failed

# View failed notification
redis-cli LRANGE notification:failed 0 -1
```

API Endpoint (if implemented)

```
GET /api/v1/notifications/queue-stats

Response:
{
  "waiting": 42,
  "processing": 3,
  "failed": 5,
  "totalProcessed": 1234
}
```

Troubleshooting

Notifications Not Processing

1. **Check queue is enabled:**

```
echo $ENABLE_NOTIFICATION_QUEUE  # Should be 'true'
```

2. **Check Redis connection:**

```
redis-cli ping  # Should return 'PONG'
```

3. **Check Service 1 is running:**

```
curl http://localhost:5000/api/v1/notification-service/health
```

4. **Check queue processor logs:**

```
# In Service 2 logs, look for:
# "🔄 Processing notification queue..."
```

High Failed Queue Count

1. **Check Service 1 authentication:**

```
# Verify INTERNAL_SERVICE_API_KEY matches in both services
```

2. **Check Service 1 Firebase credentials:**

```
# Verify FIREBASE_STRING is valid in Service 1
```

3. **Check MongoDB connection:**

```
# Service 1 needs MongoDB to fetch FCM tokens
```

4. **Manual retry failed notifications:**

```
# Move failed back to queue
redis-cli RPOPLPUSH notification:failed notification:queue
```

Migration from Old System

If you're upgrading from the old reactive-queue system:

1. **Old behavior:** HTTP first, queue on failure
2. **New behavior:** Queue first, HTTP by queue processor

No breaking changes! The API endpoints remain the same:

- `POST /send-to-user` - Still works
- `POST /send-to-users` - Still works
- `POST /send-bulk` - Still works

The only difference: notifications are now queued immediately for better reliability.

Best Practices

1. **Always enable queue in production:**

   ```
   ENABLE_NOTIFICATION_QUEUE=true
   ```

2. **Configure Redis persistence:**
   - Use AOF (appendonly yes) for durability
   - Regular RDB snapshots as backup
3. **Monitor queue length:**
   - Alert if queue length > 1000
   - Indicates Service 1 issues or high load
4. **Review failed queue daily:**
   - Check for patterns (same users failing?)
   - Retry or investigate root cause
5. **Test crash recovery regularly:**
   - Simulate crashes in staging
   - Verify notifications don't get lost

Related Documentation

- Notification Service README
- notificationQueue.Service.ts
- notificationClient.ts
- notification.Service.ts

Summary

The **queue-first architecture** provides:

- ☑ Complete crash recovery for Service 1 and Service 2
- ☑ Zero notification loss (with Redis persistence)
- ☑ Automatic retry with exponential backoff
- ☑ Async processing for better API performance
- ☑ Full monitoring and observability
- ☑ Production-ready reliability

Bottom line: All notifications are safe, even during crashes. Service restarts automatically resume processing from where they left off.