

Rate Limiting Considerations

Overview

While rate limiting is not currently implemented in the application, it is a **critical security feature** that should be added for production deployments. This document outlines recommended approaches and considerations for implementing rate limiting.

Why Rate Limiting is Important

1. **Prevent Abuse:** Protects against malicious users attempting to overwhelm the system
2. **Resource Protection:** Ensures fair resource allocation among users
3. **DoS Prevention:** Mitigates Denial of Service attacks
4. **Cost Control:** Prevents excessive usage that could increase infrastructure costs

Recommended Implementation Strategies

1. API Gateway Rate Limiting (Recommended)

Tools: Kong, AWS API Gateway, Azure API Management, NGINX

Advantages:

- Centralized rate limiting
- No application code changes required
- Easy to configure and monitor

Example Configuration (NGINX):

```
limit_req_zone $binary_remote_addr zone=general:10m rate=10r/s;
limit_req_zone $binary_remote_addr zone=message_creation:10m rate=5r/m;

server {
    location /api/ {
        limit_req zone=general burst=20 nodelay;
    }

    location /api/messages/create {
        limit_req zone=message_creation burst=2 nodelay;
    }
}
```

2. Spring Boot Rate Limiting

Library: Bucket4j with Spring Boot

Implementation:

Add dependency to `build.gradle` :

```
implementation 'com.github.vladimir-bukhtoyarov:bucket4j-core:7.6.0'
implementation 'com.github.vladimir-bukhtoyarov:bucket4j-redis:7.6.0'
```

Create Rate Limiting Configuration:

```
@Configuration
public class RateLimitConfig {

    @Bean
    public RateLimiter rateLimiter(RedissonClient redissonClient) {
        return new RedisRateLimiter(redissonClient);
    }
}
```

Create Rate Limiting Filter:

```
@Component
public class RateLimitFilter implements Filter {

    private final Map<String, Bucket> buckets = new ConcurrentHashMap<>();

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String clientId = getClientId(httpRequest);

        Bucket bucket = buckets.computeIfAbsent(clientId, k -> createBucket());

        if (bucket.tryConsume(1)) {
            chain.doFilter(request, response);
        } else {
            HttpServletResponse httpResponse = (HttpServletResponse) response;
            httpResponse.setStatus(429);
            httpResponse.getWriter().write("Too many requests");
        }
    }

    private Bucket createBucket() {
        return Bucket.builder()
            .addLimit(Bandwidth.simple(10, Duration.ofMinutes(1)))
            .build();
    }
}
```

3. NATS Rate Limiting

Since the application uses NATS for message processing, rate limiting can also be implemented at the NATS level:

Option A: NATS JetStream with Rate Limiting

- Use NATS JetStream to queue messages
- Configure consumer rate limits

Option B: Application-Level NATS Rate Limiting

```

@Service
@Slf4j
public class NatsRateLimiter {

    private final Map<String, Bucket> buckets = new ConcurrentHashMap<>();

    public boolean allowRequest(String clientId) {
        Bucket bucket = buckets.computeIfAbsent(clientId, k -> createBucket());
        return bucket.tryConsume(1);
    }

    private Bucket createBucket() {
        return Bucket.builder()
            .addLimit(Bandwidth.simple(5, Duration.ofMinutes(1)))
            .build();
    }
}

```

Recommended Rate Limits

Based on the application's use case (secret message sharing), here are recommended rate limits:

Endpoint/Operation	Rate Limit	Reason
Message Creation	5 per minute per IP	Prevents spam and abuse
Message Retrieval	10 per minute per IP	Allows multiple attempts with wrong keys
Global API	100 per minute per IP	Overall protection

Redis-Based Distributed Rate Limiting

For multi-instance deployments, use Redis to store rate limit counters:

```

@Service
public class RedisRateLimiter {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public boolean allowRequest(String clientId, int maxRequests, long windowSeconds)
    {
        String key = "rate_limit:" + clientId;
        Long currentCount = redisTemplate.opsForValue().increment(key);

        if (currentCount == 1) {
            redisTemplate.expire(key, windowSeconds, TimeUnit.SECONDS);
        }

        return currentCount <= maxRequests;
    }
}

```

Integration with Current Architecture

To add rate limiting to the current application:

1. **Add Bucket4j dependency** to `build.gradle`
2. **Create RateLimiter bean** in configuration
3. **Add rate limiting to NatsService:**

```
java
    public void createSecretMessageSubscriber(Message msg) {
        String clientId = extractClientId(msg);
        if (!rateLimiter.allowRequest(clientId)) {
            log.warn("Rate limit exceeded for client: {}", clientId);
            sendErrorResponse(msg.getReplyTo(), "Rate limit exceeded. Please try again
later.");
            return;
        }
        // ... rest of the method
    }
```

Monitoring and Alerts

Set up monitoring for:

- Number of rate-limited requests
- Rate limit hit patterns
- Potential abuse attempts

Tools: Prometheus, Grafana, ELK Stack

Configuration

Add rate limiting configuration to `application.properties` :








```
# Rate limiting configuration
app.rate-limit.enabled=true
app.rate-limit.message-creation.requests=5
app.rate-limit.message-creation.window-seconds=60
app.rate-limit.message-retrieval.requests=10
app.rate-limit.message-retrieval.window-seconds=60
app.rate-limit.global.requests=100
app.rate-limit.global.window-seconds=60
```

Security Considerations

1. **IP-Based Limiting:** Can be bypassed using proxies/VPNs
2. **Token-Based Limiting:** More secure but requires authentication
3. **Combination Approach:** Use both IP and token-based limiting
4. **Rate Limit Headers:** Return `X-RateLimit-*` headers to inform clients:
 - `X-RateLimit-Limit` : Maximum requests allowed
 - `X-RateLimit-Remaining` : Remaining requests
 - `X-RateLimit-Reset` : Time when limit resets

Production Recommendations

For production deployment:

1.  **Implement API Gateway rate limiting** (first line of defense)
2.  **Add application-level rate limiting** (defense in depth)
3.  **Use Redis for distributed rate limiting** (if multiple instances)
4.  **Monitor and alert on rate limit violations**
5.  **Regularly review and adjust limits** based on usage patterns
6.  **Implement graceful degradation** when limits are hit
7.  **Consider user tiers** (different limits for different users)

Conclusion

Rate limiting is essential for production deployments. While not currently implemented, this document provides a roadmap for adding this critical security feature. The recommended approach is to start with API Gateway rate limiting and then add application-level rate limiting for defense in depth.