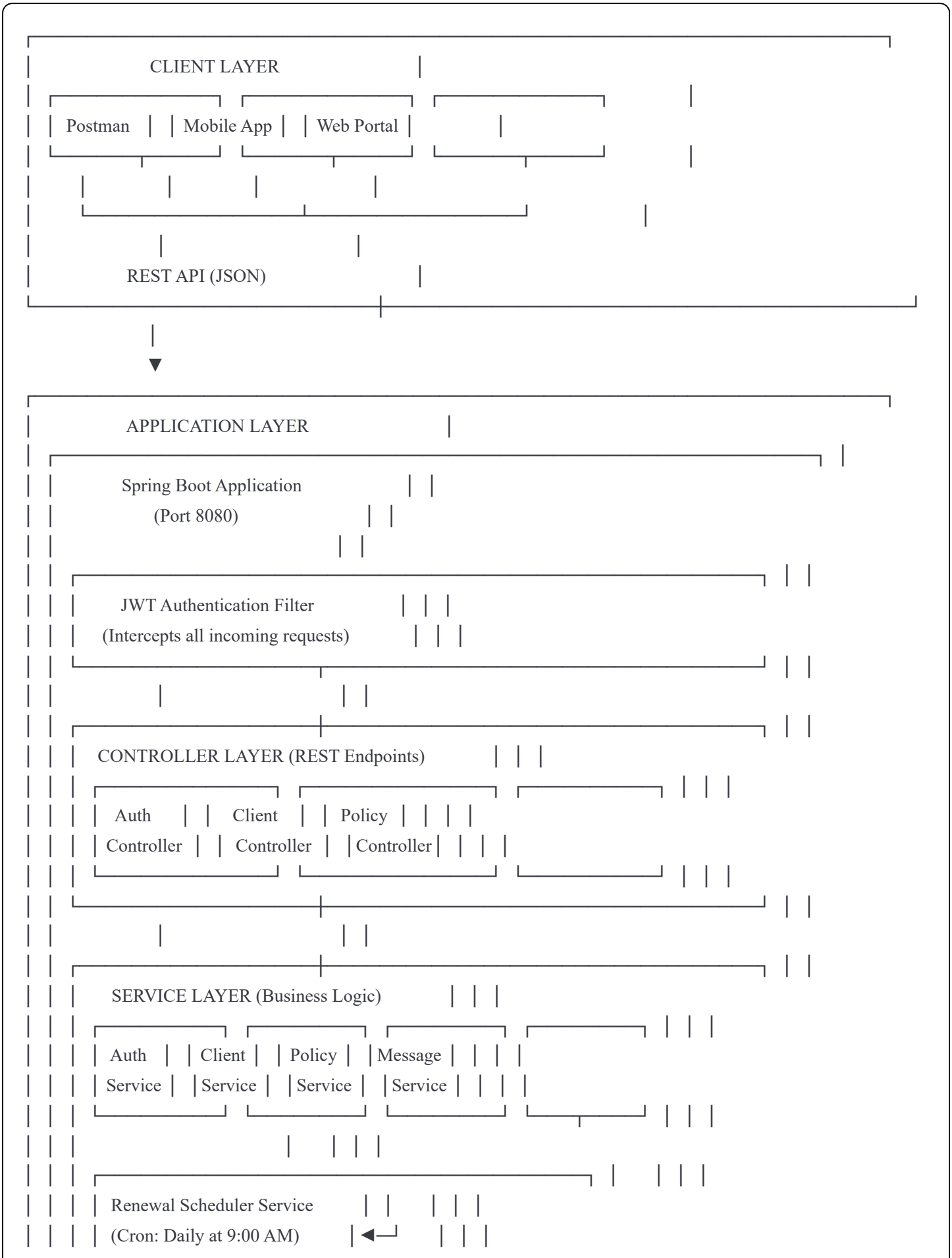
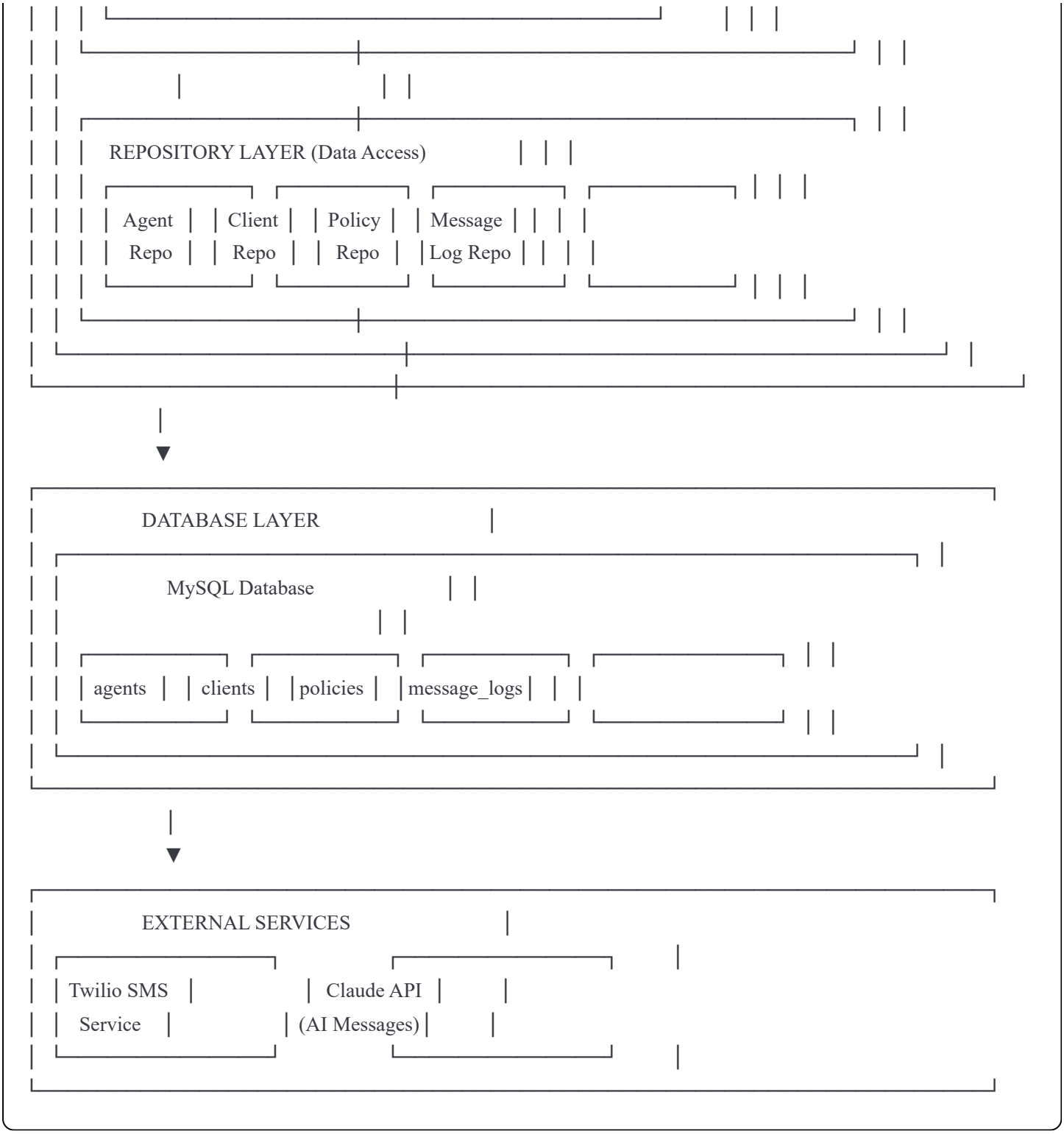
Renew AI - System Architecture Documentation

1. High-Level Architecture





---

2. Request Flow Diagrams

2.1 Authentication Flow

```
User → POST /api/auth/login
  |
  ├→ AuthController.login()
  |
  ├→ AuthService.login()
  |   |
```

```
  │   ├──→ AgentRepository.findByUsername()
  │   │   │
  │   │   └──→ MySQL: SELECT * FROM agents WHERE username = ?
  │   │
  │   ├──→ BCrypt.matches(password, hashedPassword)
  │   │
  │   └──→ JwtUtil.generateToken(username, agentId)
  │
  ├──→ Return LoginResponse (with JWT token)
  │
User ←─ { token: "eyJhbGc...", agentId: 1, ... }
```

2.2 Protected Endpoint Flow

```
User ──→ GET /api/clients (with JWT in header)
  │
  ├──→ JwtAuthenticationFilter.doFilterInternal()
  │   │
  │   ├──→ Extract "Bearer token" from Authorization header
  │   │
  │   ├──→ JwtUtil.extractUsername(token)
  │   │
  │   ├──→ JwtUtil.validateToken(token, username)
  │   │
  │   └──→ Set Authentication in SecurityContext
  │
  ├──→ ClientController.getMyClients(Authentication)
  │
  ├──→ ClientService.getClientsByAgent(username)
  │   │
  │   ├──→ AgentRepository.findByUsername()
  │   │
  │   └──→ ClientRepository.findByAgent(agent)
  │
User ←─ [ { id: 1, fullName: "..." }, ... ]
```

2.3 Scheduler Flow (Daily at 9:00 AM)

```
System Clock: 9:00 AM
  │
  ├──→ Spring Scheduler triggers
  │
  ├──→ RenewalSchedulerService.checkExpiringPoliciesAndSendReminders()
  │   │
  │   ├──→ Calculate dates:
```

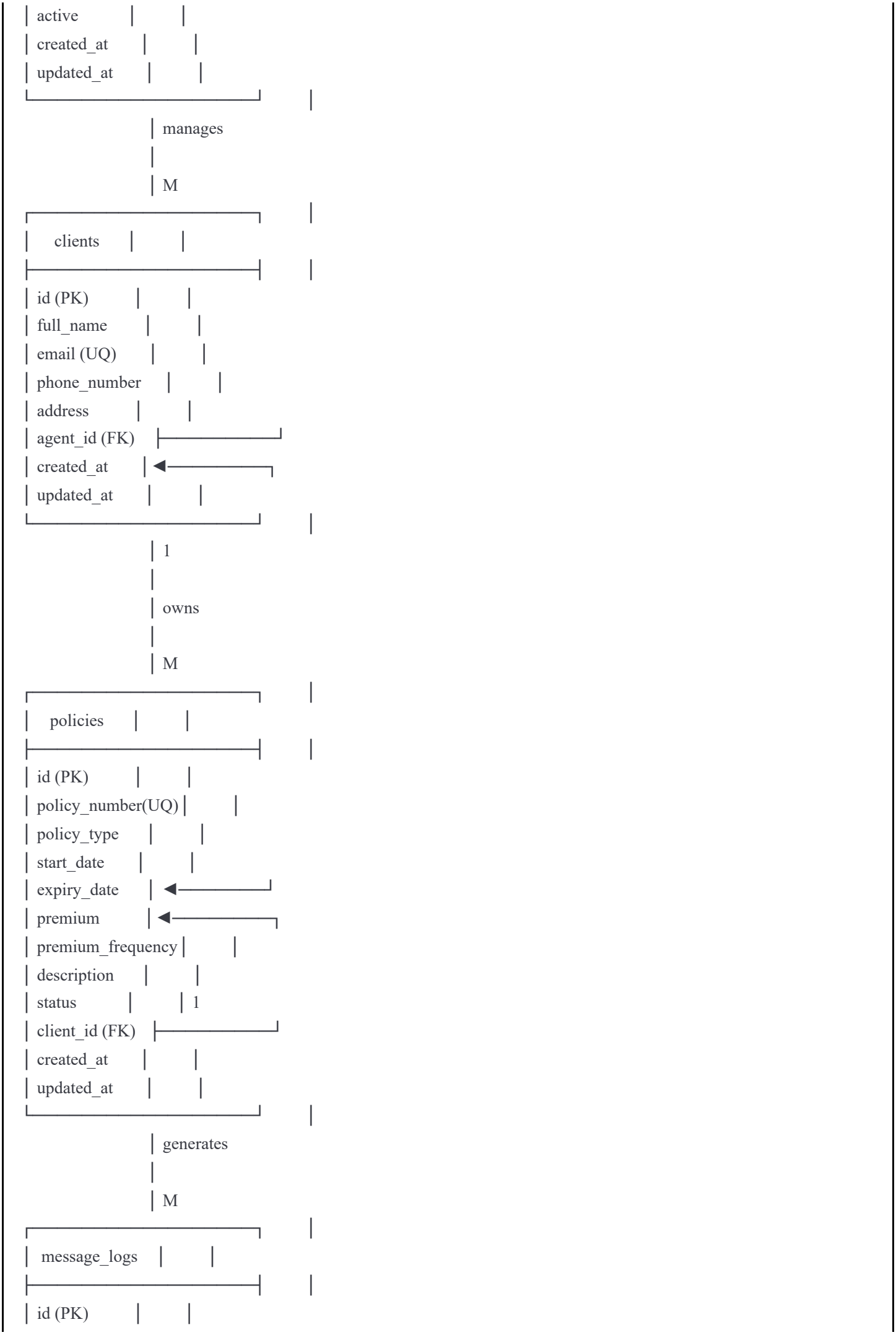
```
│   │   - sevenDaysLater = today + 7 days
│   │   - threeDaysLater = today + 3 days
│   │
│   ├──→ PolicyService.getPoliciesExpiringOn(sevenDaysLater)
│   │   │
│   │   └──→ MySQL: SELECT * FROM policies WHERE expiry_date = ?
│   │
│   ├──→ For each policy:
│   │   │
│   │   ├──→ MessageService.sendRenewalReminder(policy, "SEVEN_DAYS", 7)
│   │   │   │
│   │   │   ├──→ Check if already sent:
│   │   │   │    MessageLogRepository.existsByPolicyIdAndReminderType()
│   │   │   │
│   │   │   ├──→ If not sent:
│   │   │   │   │
│   │   │   │   ├──→ MessageService.generateRenewalMessage()
│   │   │   │   │    (AI integration point - can call Claude API)
│   │   │   │   │
│   │   │   │   ├──→ Send SMS via Twilio/Mock
│   │   │   │   │
│   │   │   │   └──→ Save to message_logs table
│   │   │   │
│   │   │   └──→ Return true/false
│   │   │
│   │   └──→ Log result
│   │
│   └──→ Repeat for threeDaysLater
│
└──→ Log completion and statistics
```

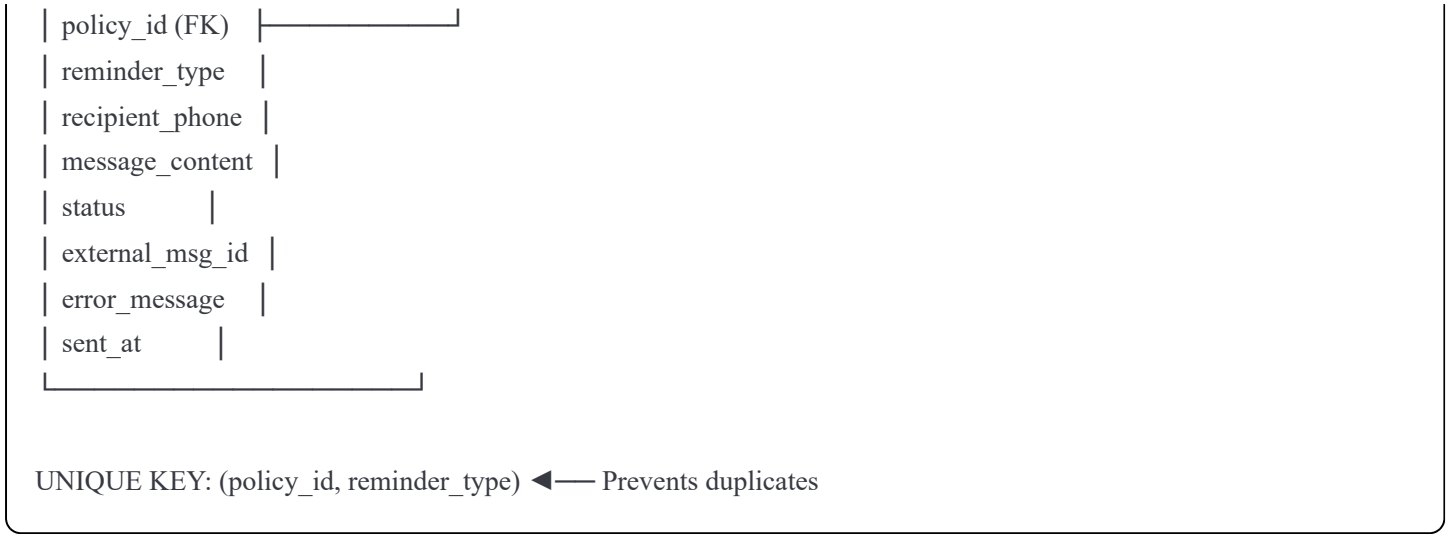
---

3. Database Schema

3.1 Entity Relationship Diagram

```
┌──────────────────────────┐
│      agents              │
├──────────────────────────┤
│ id (PK)                  │
│ username (UQ)            │ ◄──────────┐
│ email (UQ)               │            │
│ password (BCrypt)        │            │
│ full_name                │            │ 1
│ phone_number             │            │
```


active
created_at
updated_at
manages
M
clients
id (PK)
full_name
email (UQ)
phone_number
address
agent_id (FK)
created_at
updated_at
1
owns
M
policies
id (PK)
policy_number(UQ)
policy_type
start_date
expiry_date
premium
premium_frequency
description
status
1
client_id (FK)
created_at
updated_at
generates
M
message_logs
id (PK)



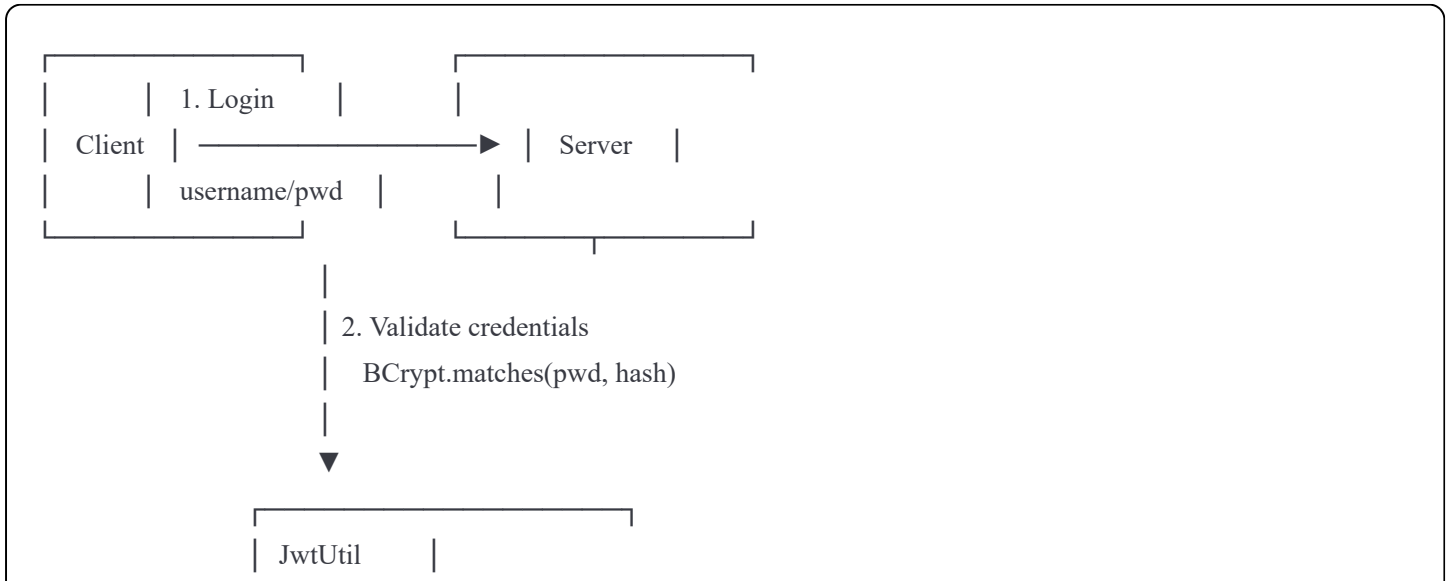
3.2 Key Constraints

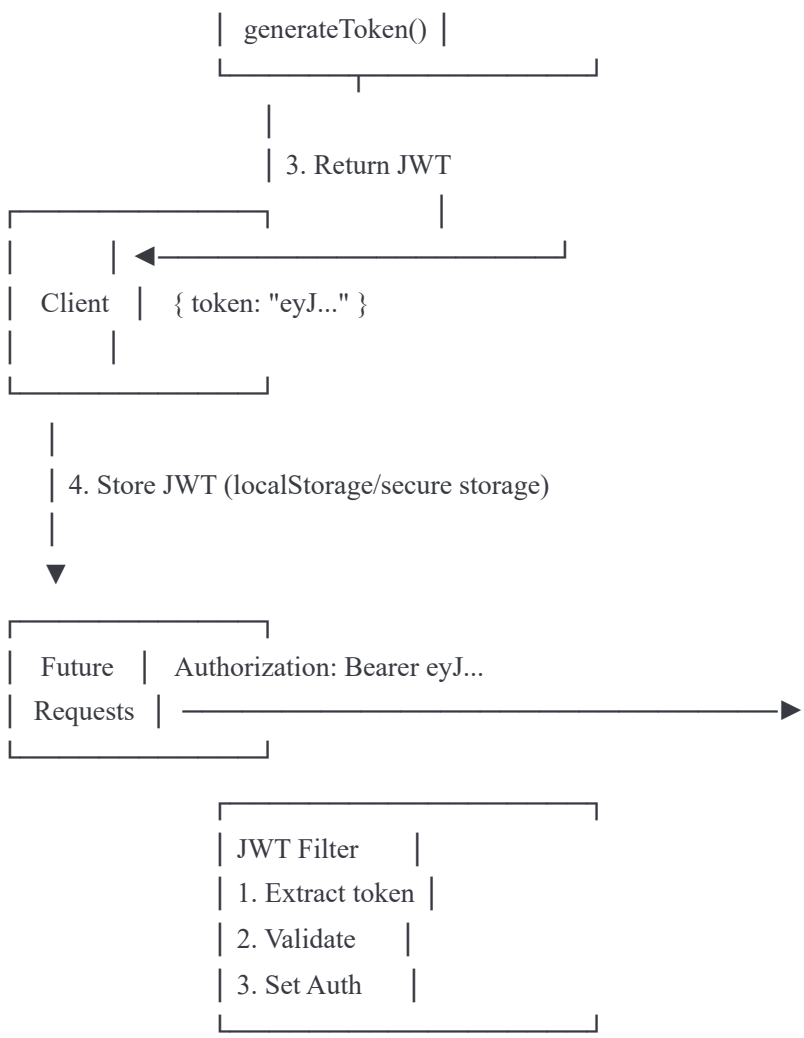
1. **Primary Keys:** Auto-increment BIGINT on all tables
2. **Foreign Keys:** Cascade delete (if agent deleted, clients deleted too)
3. **Unique Constraints:**
   - agents: username, email
   - clients: email
   - policies: policy_number
   - message_logs: (policy_id, reminder_type) - **Prevents duplicate reminders**
4. **Indexes:**
   - policies.expiry_date - For fast scheduler queries
   - policies.status - For filtering active policies

---

4. Security Architecture

4.1 JWT Flow





4.2 Password Security

```
Registration/Update:
  User Password: "agent123"
        │
        ▼
  BCrypt.encode(password, strength=12)
        │
        ▼
  Hashed: "$2a$12$LQv3c1yqBWVHxkd0LHAkCOYz6TtxMQ..."
        │
        ▼
  Store in database

Login:
  User Password: "agent123"
        │
        ▼
  BCrypt.matches(input, storedHash)
        │
        ├─ Match → Generate JWT
```

```
    │
    └─ No Match → Throw exception
```

---

5. Scheduler Architecture

5.1 Why Spring Scheduler?

Feature	Spring Scheduler	Message Queue	Manual Execution
Setup Complexity	✅ Simple	❌ Complex	✅ Simple
Time-based	✅ Built-in	⚠️ Requires wrapper	❌ Manual
Scalability	⚠️ Single instance	✅ Distributed	❌ Not scalable
Reliability	✅ Auto-retry	✅ High	❌ Human error
Cost	✅ Free	⚠️ Additional service	✅ Free
Use Case	⚠️ Time-based	✅ Event-based	❌ Not recommended

Verdict: For time-based, single-instance reminders, Spring Scheduler is perfect.

5.2 Scheduler State Machine

```
IDLE (waiting for 9:00 AM)
  │
  │ Cron triggers: 0 0 9 * * ?
  │
  ▼
EXECUTING
  │
  ├─▶ Find policies expiring in 7 days
  │   │
  │   ├─▶ For each policy:
  │   │   │
  │   │   ├─▶ Check message_logs (duplicate?)
  │   │   │   │
  │   │   │   ├─▶ Already sent → SKIP
  │   │   │   │
  │   │   │   └─▶ Not sent:
  │   │   │       │
  │   │   │       ├─▶ Generate message
  │   │   │       ├─▶ Send SMS
  │   │   │       └─▶ Log in database
```

```
│   │   │
│   │   └─▶ NEXT POLICY
│   │
│   └─▶ Log statistics
│
├─▶ Find policies expiring in 3 days
│   (same process as above)
│
└─▶ COMPLETED
    │
    │ Wait 24 hours
    │
    └─▶ Back to IDLE
```

---

6. AI Integration Architecture

6.1 Current vs AI-Enhanced Message Generation

Current (Template-based):

```
Policy Data → Template String Format → Static Message
```

AI-Enhanced:

```
Policy Data → Claude API Prompt → Personalized Message → Fallback to Template (if API fails)
```

6.2 AI Integration Point in Code

File: `MessageService.java`

```java

```

```
public String generateRenewalMessage(Policy policy, int daysUntilExpiry) {
    // Current: Template-based
    String message = String.format(
        "Dear %s, your %s policy expires in %d days...",
        clientName, policyType, daysUntilExpiry
    );

    // AI Enhancement (pseudocode):
    try {
        // 1. Prepare context
        String prompt = buildPrompt(policy, daysUntilExpiry);

        // 2. Call Claude API
        ClaudeResponse response = claudeClient.sendMessage(prompt);

        // 3. Extract message
        String aiMessage = response.getContent().get(0).getText();

        // 4. Validate (length, tone, required info)
        if (isValidMessage(aiMessage)) {
            return aiMessage;
        }
    } catch (Exception e) {
        logger.error("AI generation failed", e);
    }

    // 5. Fallback to template
    return message;
}
```

6.3 AI Design Principles

✅ **DO:**

- Use AI for message personalization
- Keep business logic in Java code
- Have fallback templates
- Validate AI output
- Log AI vs template usage

❌ **DON'T:**

- Let AI decide when to send
- Let AI decide who to send to
- Rely solely on AI (no fallback)

- Let AI modify business rules
- Expose sensitive data to AI

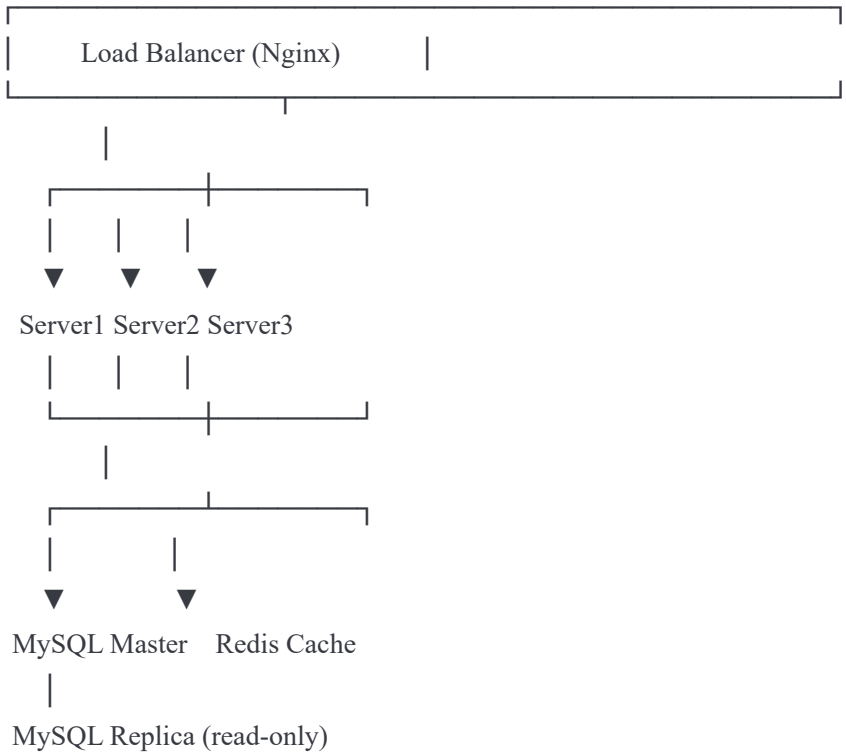
---

7. Scalability Considerations

7.1 Current Architecture (Single Instance)

```
Single Server
  ├─ Spring Boot App
  ├─ Scheduler (runs once)
  ├─ MySQL (single instance)
  └─ Handles ~1000 policies/day
```

7.2 Scaled Architecture (Production)



Enhancements:

1. **Scheduler:** Use distributed lock (Redis) to prevent duplicate executions
2. **Database:** Master-slave replication (writes to master, reads from replica)
3. **Caching:** Redis for frequently accessed policies
4. **Message Queue:** RabbitMQ for high-volume SMS (100k+ policies)
5. **Monitoring:** Prometheus + Grafana for metrics

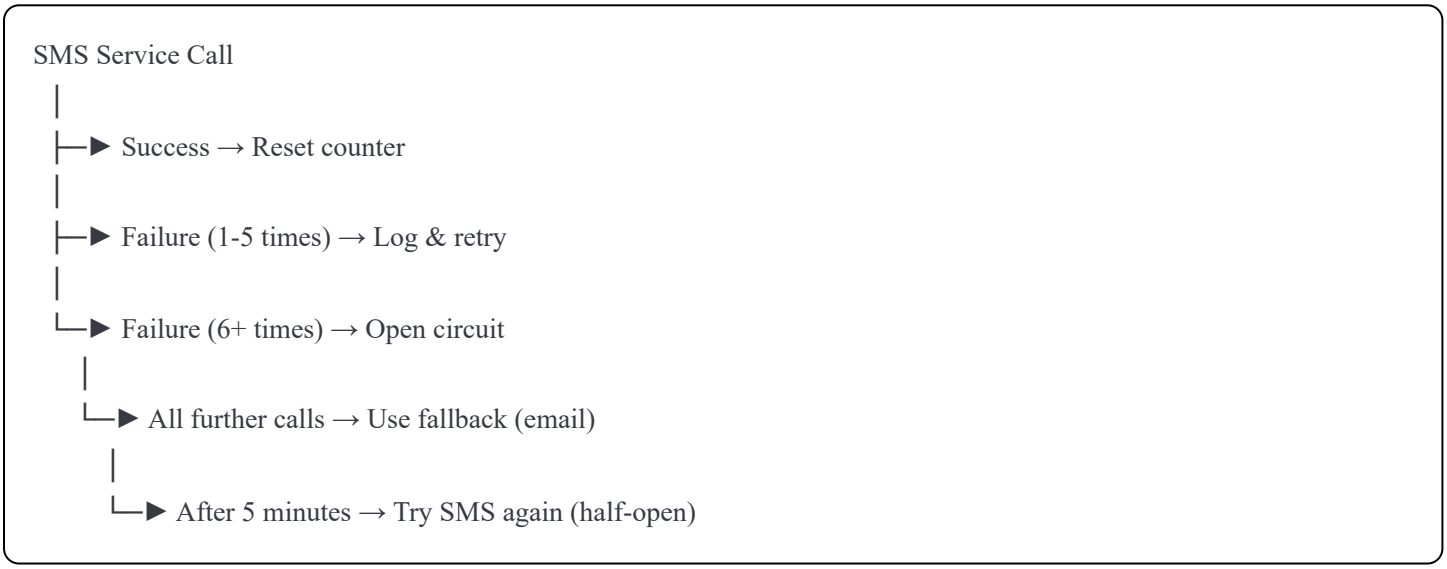
---

8. Error Handling & Resilience

8.1 Scheduler Error Scenarios

Error	Handling	Impact
Database down	Try-catch, log error, wait for next run	Messages delayed 24 hours
SMS service down	Mark as FAILED, retry next run	Messages delayed but logged
Invalid phone number	Log error, skip policy	Single policy skipped
Duplicate constraint violation	Caught by DB, skip silently	No impact (already sent)

8.2 Circuit Breaker Pattern (Future Enhancement)



---

9. Monitoring & Observability

Key Metrics to Track

1. **Scheduler Execution:**
    - Last run timestamp
    - Policies processed
    - Messages sent/skipped/failed
    - Execution time
2. **API Performance:**
    - Request count
    - Response time
    - Error rate

- JWT validation failures

3. **Database:**
    - Connection pool usage
    - Query execution time
    - Table sizes
4. **SMS Service:**
    - Delivery success rate
    - Average delivery time
    - Cost per message

---

10. Deployment Architecture

Development

```
Local Machine → Maven → Embedded Tomcat → MySQL (localhost)
```

Production

```
Git Push → Jenkins CI/CD → Docker Build → AWS ECS/EC2
                                          |
                                          ├─ RDS MySQL
                                          ├─ ElastiCache Redis
                                          └─ CloudWatch Logs
```

This architecture is designed to be:

- ✅ Interview-friendly (explains WHY, not just HOW)
- ✅ Production-ready (with scaling considerations)
- ✅ Maintainable (clean separation of concerns)
- ✅ Extensible (AI integration point, microservices-ready)