Table of Contents

# MLM 시스템 설계 문서

**버전**: 3.0 **작성일**: 2025년 10월 3일 **최종 수정**: 2025년 10월 3일 (지급 규칙 변경 반영) **시스템명**: NanumPay MLM Management System

## 목차

1. [시스템 개요](#X8027de47315d76e77f1153ee70e622ea2bc586b)
2. [데이터베이스 설계](#X8600f41c7e4d698ff8a8bd8c9034de0ef938928)
3. [등급 계산 시스템](#X66fe59dbc4af6e4a68e02f891507a740c0b3f7e)
4. [매출 및 지급 시스템](#Xce818992c7c30a0b3d039c451061393c90eae53)
5. [시스템 아키텍처](#X1e158012178f91247ccb40b360d269f298d4aa3)
6. [운영 지침](#X2b3688ab3842a6e72bc531ab4778521f9f7dd95)

## 1. 시스템 개요

### 1.1 시스템 목적

MLM(Multi-Level Marketing) 비즈니스 모델을 지원하는 용역자 관리 및 수익 분배 시스템

### 1.2 핵심 기능

* **이진트리 구조** 기반 조직 관리
* **8단계 등급 시스템** (F1~F8)
* **자동 매출 계산** 및 **10회 분할 지급**
* **누적식 수익 배분** 방식

### 1.3 시스템 특징

* ✅ 완전 자동화된 등급 계산
* ✅ 투명한 수익 분배 구조
* ✅ 실시간 트리 구조 시각화
* ✅ 배치 처리 최적화

## 2. 데이터베이스 설계

### 2.1 컬렉션 구조

#### 2.1.1 기본 컬렉션 (MongoDB)

##### **admins 컬렉션**

{  
 \_id: ObjectId,  
 loginId: String, // 관리자 로그인 ID  
 passwordHash: String, // bcrypt 해시  
 name: String,  
 email: String,  
 phone: String,  
 permissions: [String], // 권한 배열  
 isActive: Boolean,  
 createdAt: Date  
}

* **시스템 관리자 계정** (로그인 및 권한 관리 전용)
* 이진트리 구조와 무관 (별도 관리)
* 용역자 트리는 users 컬렉션에서 독립적으로 관리

##### **users 컬렉션**

{  
 \_id: ObjectId,  
 loginId: String, // 한글이름 (중복시 A,B,C)  
 passwordHash: String, // 전화번호 뒷4자리 해시  
 name: String,  
 phone: String,  
 bank: String,  
 accountNumber: String,  
 parentId: String, // 부모 용역자 loginId (루트는 null)  
 leftChildId: String, // 왼쪽 자식 loginId  
 rightChildId: String, // 오른쪽 자식 loginId  
 position: "L" | "R", // 부모 기준 위치 (루트는 null)  
 grade: String, // F1~F8  
 createdAt: Date  
}

**중요:** - parentId가 null인 용역자 = **트리 최상위 루트 노드 (시스템 전체 1개만)** - 모든 용역자는 users 컬렉션에서 이진트리 구조로 관리 - admins 컬렉션과는 완전히 독립적

##### **monthlyrevenues 컬렉션**

{  
 \_id: ObjectId,  
 month: "2024-09",  
 totalRevenue: Number, // 총매출액  
 userCount: Number, // 신규 등록 인원  
 gradeDistribution: {  
 F1: Number,  
 F2: Number,  
 // ... F8까지  
 },  
 createdAt: Date  
}

##### **userpaymentplans 컬렉션**

{  
 \_id: ObjectId,  
 userId: String, // 용역자 loginId  
 revenueMonth: "2024-09", // 매출 발생월  
 grade: String, // 매출 시점 등급  
 totalAmount: Number, // 총 지급액  
 installmentAmount: Number, // 회차당 지급액  
 payments: [{  
 installment: Number, // 회차 (1~10)  
 scheduledWeek: "2024-10-W1",  
 amount: Number,  
 status: "pending" | "paid",  
 paidDate: Date,  
 transactionId: String  
 }],  
 createdAt: Date  
}

##### **weeklypayments 컬렉션**

{  
 \_id: ObjectId,  
 week: "2024-10-W1",  
 userId: String,  
 totalAmount: Number,  
 withholdingTax: Number, // 원천징수 3.3%  
 netAmount: Number, // 실지급액  
 details: [{  
 revenueMonth: String,  
 installment: Number,  
 amount: Number  
 }],  
 createdAt: Date  
}

### 2.2 계층 구조

[admins 컬렉션] [users 컬렉션 - 이진트리]  
 관리자  
 (시스템 로그인) 용역자A(F6)  
 [parentId=null, 루트 1개만]  
 |  
 +--+--+  
 | |  
 B(F4) C(F3)  
 | |  
 +--+--+ +--+--+  
 | | | |  
 D(F2) E(F2) F(F1) G(F1)  
 <- parentId = 부모 loginId

**구조 설명:** - **admins**: 시스템 관리자 (로그인 전용, 트리 무관) - **users**: 용역자 이진트리 - parentId = null: **전체 시스템에 단 1개만 존재하는 루트 노드** - parentId = loginId: 부모 용역자의 loginId 참조

### 2.3 등록 프로세스

#### 2.3.0 등록 방식 개요

**용역자 등록 방법:** 1. **엑셀 일괄 등록**: API /api/admin/users/bulk 2. **개별 등록**: API /api/admin/users/register

**공통 필수 항목:** - 성명 (name) - 연락처 (phone) - 은행 (bank) - 계좌번호 (accountNumber) - 판매인/추천인 (salesperson) - 가입일자 (date)

**공통 자동 처리:** - loginId: 성명 소문자 변환 (중복 시 A, B, C 추가) - password: 전화번호 뒷 4자리 (기본 1234) - position: 판매인의 L→R 우선 배치 - grade: 초기 F1, 이후 자동 재계산

#### 2.3.1 엑셀 일괄 등록

**헤더 형식:**

순번, 날짜, 성명, 연락처, 주민번호, 은행, 계좌번호,  
판매인, 연락처, 설계사, 연락처, 보험상품명, 보험회사, 지사

**헤더 매핑 (유연한 형식 지원):**

// 다양한 헤더명 지원  
성명: ['성명', '이름', 'name', '\_\_EMPTY\_1']  
연락처: ['연락처', '전화번호', 'phone', '\_\_EMPTY\_2']  
날짜: ['날짜', 'date', '\_\_EMPTY']  
판매인: ['판매인', '추천인', 'salesperson', '\_\_EMPTY\_6']  
  
// \_\_EMPTY 형식 처리 (xlsx 라이브러리)  
\_\_EMPTY: 날짜  
\_\_EMPTY\_1: 성명  
\_\_EMPTY\_2: 연락처  
...

**헤더 행 처리:**

// 첫 번째 행이 헤더인 경우 자동 건너뛰기  
if (userData['용 역 자 관 리 명 부'] === '순번') {  
 continue; // 헤더 행 스킵  
}

#### 2.3.2 2단계 등록 프로세스 (핵심)

**1단계: 사용자 등록 (부모 관계 없이)**

for (const userData of users) {  
 // 1. loginId 자동 생성  
 let loginId = name.toLowerCase();  
 while (await User.exists({ loginId })) {  
 counter++;  
 loginId = name.toLowerCase() + String.fromCharCode(64 + counter); // A, B, C...  
 }  
  
 // 2. 비밀번호 생성  
 const phoneDigits = phone.replace(/[^0-9]/g, '');  
 const password = phoneDigits.length >= 4   
 ? phoneDigits.slice(-4) // 전화번호 뒷 4자리  
 : '1234'; // 기본값  
  
 // 3. 시퀀스 번호 할당 (등록 순서 추적)  
 currentSequence++;  
  
 // 4. createdAt 설정 (매출 귀속 월 결정)  
 let createdAt;  
 if (dateValue) {  
 // Excel 날짜 숫자 형식  
 if (!isNaN(dateValue)) {  
 const excelDate = parseInt(dateValue);  
 createdAt = new Date((excelDate - 25569) \* 86400 \* 1000);  
 } else {  
 // 문자열 날짜  
 createdAt = new Date(dateValue);  
 }  
 } else {  
 createdAt = new Date(); // 오늘 날짜  
 }  
  
 // 5. 사용자 생성 (parentId 없이, 판매인 정보만 저장)  
 const newUser = new User({  
 loginId,  
 passwordHash: await bcrypt.hash(password, 10),  
 name,  
 phone,  
 salesperson, // 나중에 부모 찾기용  
 grade: 'F1', // 초기 등급  
 sequence: currentSequence,  
 createdAt: createdAt // 매출 귀속 월 결정  
 });  
  
 await newUser.save();  
 registeredUsers.set(loginId, { user: newUser, salesperson, name });  
}

**2단계: 부모-자식 관계 설정 (엑셀 순서대로)**

**MLM 검증 룰:**

// 1. 시스템 전체 루트 노드 검증 (전체 시스템에 루트는 1개만)  
const rootCandidates = registeredUsers.filter(info =>  
 info.salesperson === '-' || !info.salesperson  
);  
  
if (rootCandidates.length > 1) {  
 throw new Error('엑셀 파일 내 루트 노드는 1명만 허용됩니다');  
}  
  
// 기존 시스템에 루트 존재 확인  
const existingRoot = await User.findOne({ parentId: null });  
  
if (rootCandidates.length === 1 && existingRoot) {  
 throw new Error(`시스템에 이미 루트 노드가 존재합니다: ${existingRoot.name} (${existingRoot.loginId})`);  
}  
  
// 2. 자기 자신 판매인 방지  
if (info.salesperson === info.name) {  
 throw new Error('자기 자신을 판매인으로 등록할 수 없습니다');  
}

**중요**: 전체 시스템에서 parentId = null인 루트 노드는 **단 1개만** 허용됩니다.

**판매인 검색 우선순위:**

for (const orderInfo of usersByOrder) {  
 const info = registeredUsers.get(orderInfo.loginId);  
   
 // 루트 노드는 부모 없이 그대로 둠  
 if (info.salesperson === '-' || !info.salesperson) {  
 continue;  
 }  
  
 // 판매인 검색 (우선순위)  
 let parentUser = null;  
  
 // 우선순위 1: 기존 DB 사용자  
 parentUser = await User.findOne({  
 $or: [  
 { name: info.salesperson },  
 { loginId: info.salesperson.toLowerCase() }  
 ],  
 type: 'user'  
 });  
  
 // 우선순위 2: 방금 등록한 사용자  
 if (!parentUser) {  
 for (const [regLoginId, regInfo] of registeredUsers) {  
 if (regInfo.name === info.salesperson) {  
 parentUser = regInfo.user;  
 break;  
 }  
 }  
 }  
  
 // 판매인을 찾을 수 없으면 실패  
 if (!parentUser) {  
 failedUsers.push(info.name);  
 results.errors.push(`❌ ${info.name}: 판매인 '${info.salesperson}'을(를) 찾을 수 없습니다.`);  
 continue;  
 }  
  
 // L/R 위치 결정  
 const leftChild = await User.findOne({   
 parentId: parentUser.loginId,   
 position: 'L'   
 });  
 const rightChild = await User.findOne({   
 parentId: parentUser.loginId,   
 position: 'R'   
 });  
  
 let position = null;  
 if (!leftChild) {  
 position = 'L'; // 왼쪽 우선  
 } else if (!rightChild) {  
 position = 'R'; // 왼쪽 차있으면 오른쪽  
 }  
  
 if (position) {  
 // 부모-자식 관계 설정  
 await User.findOneAndUpdate(  
 { loginId: info.user.loginId },  
 { parentId: parentUser.loginId, position }  
 );  
  
 // 부모의 자식 참조 업데이트  
 const updateField = position === 'L' ? 'leftChildId' : 'rightChildId';  
 await User.findOneAndUpdate(  
 { loginId: parentUser.loginId },  
 { [updateField]: info.user.loginId }  
 );  
  
 // 부모 등급 재계산  
 await updateParentGrade(parentUser.loginId);  
 } else {  
 // 좌우 자리 모두 찬 경우  
 failedUsers.push(info.name);  
 results.alerts.push({  
 type: 'warning',  
 message: `${info.salesperson}님의 좌우 자리가 모두 찼습니다. ${info.name}님은 수동으로 배치해 주세요.`,  
 parent: info.salesperson,  
 user: info.name  
 });  
 }  
}  
  
// 실패한 사용자 삭제  
for (const userName of failedUsers) {  
 const userInfo = registeredUsers.find(info => info.name === userName);  
 if (userInfo) {  
 await User.findByIdAndDelete(userInfo.user.\_id);  
 }  
}

#### 2.3.3 개별 등록

**API 엔드포인트:** /api/admin/users/register

**요청 본문:**

{  
 name: String, // 성명 (필수)  
 phone: String, // 연락처 (필수)  
 bank: String, // 은행 (필수)  
 accountNumber: String, // 계좌번호 (필수)  
 salesperson: String, // 판매인 이름 (필수, "-"는 루트 노드)  
 date: String, // 가입일자 (선택, 기본값: 오늘)  
}

**처리 과정:**

async function registerUser(userData) {  
 // 1. loginId 생성 (중복 체크)  
 let loginId = userData.name.toLowerCase();  
 let counter = 0;  
 while (await User.exists({ loginId })) {  
 counter++;  
 loginId = userData.name.toLowerCase() + String.fromCharCode(64 + counter);  
 }  
   
 // 2. 비밀번호 생성  
 const phoneDigits = userData.phone.replace(/[^0-9]/g, '');  
 const password = phoneDigits.length >= 4 ? phoneDigits.slice(-4) : '1234';  
   
 // 3. 가입일자 처리  
 const createdAt = userData.date ? new Date(userData.date) : new Date();  
   
 // 4. 판매인 검색 및 루트 노드 검증  
 let parentUser = null;  
  
 if (userData.salesperson === '-' || !userData.salesperson) {  
 // 루트 노드 등록 시도 - 시스템 전체에 1개만 허용  
 const existingRoot = await User.findOne({ parentId: null });  
 if (existingRoot) {  
 throw new Error(`시스템에 이미 루트 노드가 존재합니다: ${existingRoot.name} (${existingRoot.loginId})`);  
 }  
 } else {  
 // 일반 노드 - 판매인 검색  
 parentUser = await User.findOne({  
 $or: [  
 { name: userData.salesperson },  
 { loginId: userData.salesperson.toLowerCase() }  
 ]  
 });  
  
 if (!parentUser) {  
 throw new Error('판매인을 찾을 수 없습니다');  
 }  
 }  
   
 // 5. L/R 위치 결정  
 let position = null;  
 if (parentUser) {  
 const leftChild = await User.findOne({   
 parentId: parentUser.loginId,   
 position: 'L'   
 });  
 const rightChild = await User.findOne({   
 parentId: parentUser.loginId,   
 position: 'R'   
 });  
   
 if (!leftChild) position = 'L';  
 else if (!rightChild) position = 'R';  
 else throw new Error('판매인의 좌우 자리가 모두 찼습니다');  
 }  
   
 // 6. 사용자 생성  
 const newUser = new User({  
 loginId,  
 passwordHash: await bcrypt.hash(password, 10),  
 name: userData.name,  
 phone: userData.phone,  
 bank: userData.bank,  
 accountNumber: userData.accountNumber,  
 parentId: parentUser ? parentUser.loginId : null,  
 position: position,  
 grade: 'F1',  
 createdAt: createdAt  
 });  
   
 await newUser.save();  
   
 // 7. 부모의 자식 참조 업데이트  
 if (parentUser && position) {  
 const updateField = position === 'L' ? 'leftChildId' : 'rightChildId';  
 await User.findOneAndUpdate(  
 { loginId: parentUser.loginId },  
 { [updateField]: newUser.loginId }  
 );  
 }  
   
 // 8. 등급 재계산 및 매출 계산  
 await processNewUsers([newUser.loginId]);  
   
 return newUser;  
}

#### 2.3.4 배치 처리 최적화

// 효율적 배치 처리  
const allUsers = await User.find().lean();  
const gradeUpdates = calculateAllGrades(allUsers);  
await User.bulkWrite(gradeUpdates);  
await UserPaymentPlan.insertMany(plans);

#### 2.3.5 자동 배치 처리 트리거

**실행 시점:** 1. **엑셀 일괄 등록 시** (/api/admin/users/bulk) - 모든 사용자 등록 완료 후 자동 실행

1. **개별 등록 시** (/api/admin/users/register)
   * 1명 등록 후 자동 실행
2. **수동 재계산 시** (/api/admin/revenue/calculate)
   * 관리자가 명시적으로 실행

**처리 순서:**

// BatchProcessor.processNewUsers() 실행  
async function processNewUsers(userIds) {  
 // 1. 전체 등급 재계산 (리프부터 상향식)  
 await recalculateAllGrades();  
  
 // 2. 월별 그룹화 (createdAt 기준)  
 const usersByMonth = new Map();  
 for (const user of users) {  
 const year = user.createdAt.getFullYear();  
 const month = user.createdAt.getMonth() + 1;  
 const monthKey = `${year}-${String(month).padStart(2, '0')}`;  
   
 if (!usersByMonth.has(monthKey)) {  
 usersByMonth.set(monthKey, []);  
 }  
 usersByMonth.get(monthKey).push(user);  
 }  
  
 // 3. 각 월별 매출 계산  
 for (const [monthKey, users] of usersByMonth) {  
 const [year, month] = monthKey.split('-').map(Number);  
 await calculateMonthlyRevenueForMonth(year, month);  
 }  
  
 // 4. 지급 스케줄 생성  
 await createPaymentSchedules();  
  
 // 5. 개인별 지급 계획 생성  
 await createUserPaymentPlans();  
}

#### 2.3.6 매출 귀속 월 결정 규칙

**기본 원칙:** - createdAt 필드의 년/월을 기준으로 매출 귀속 월 결정 - 엑셀 “날짜” 컬럼 → createdAt 매핑

**날짜 처리 우선순위:**

// 1. Excel 숫자 형식 (예: 45231 = 2023-10-15)  
if (!isNaN(dateValue)) {  
 const excelDate = parseInt(dateValue);  
 createdAt = new Date((excelDate - 25569) \* 86400 \* 1000);  
}  
  
// 2. 문자열 형식 (예: '2025-07-01', '2025/07/01', '20250701')  
else {  
 createdAt = new Date(dateValue);  
}  
  
// 3. 날짜가 없거나 유효하지 않으면 오늘 날짜  
if (!dateValue || isNaN(createdAt.getTime())) {  
 createdAt = new Date();  
}

**매출 계산 시 월별 그룹화:**

// 예시: 하나의 엑셀에 여러 달 데이터 포함 가능  
// 2025년 7월: 5명  
// 2025년 8월: 3명  
// 2025년 9월: 2명  
  
→ 각각 별도 MonthlyRevenue 생성  
 - 2025-07: 5명 × 100만원 = 500만원  
 - 2025-08: 3명 × 100만원 = 300만원  
 - 2025-09: 2명 × 100만원 = 200만원

## 3. 등급 계산 시스템

### 3.1 등급 정의

| 등급 | 조건 | 설명 |
| --- | --- | --- |
| **F1** | 자식 0~1개 | 리프 또는 불완전 노드 |
| **F2** | 좌우 자식 모두 | 완전 이진 노드 |
| **F3** | 좌우 각 F2 1개씩 | F2 균형 보유 |
| **F4** | 좌우 각 F3 1개씩 | F3 균형 보유 |
| **F5** | F4 총 3개 이상 (최소 1:2 또는 2:1) | F4 비대칭 허용 |
| **F6** | F5 총 3개 이상 (최소 1:2 또는 2:1) | F5 비대칭 허용 |
| **F7** | F6 총 3개 이상 (최소 1:2 또는 2:1) | F6 비대칭 허용 |
| **F8** | F7 총 3개 이상 (최소 1:2 또는 2:1) | 최고 등급 |

### 3.2 계산 알고리즘

async function calculateGrade(userId) {  
 // 1. 자식 확인  
 const leftChild = await User.findOne({ parentId: userId, position: 'L' });  
 const rightChild = await User.findOne({ parentId: userId, position: 'R' });  
  
 // 2. F1: 자식 불완전  
 if (!leftChild || !rightChild) return 'F1';  
  
 // 3. 서브트리 등급 수집  
 const leftGrades = await collectSubtreeGrades(leftChild.loginId);  
 const rightGrades = await collectSubtreeGrades(rightChild.loginId);  
  
 // 4. 등급 결정 로직  
 if (leftGrades.F2 >= 1 && rightGrades.F2 >= 1) {  
 if (leftGrades.F3 >= 1 && rightGrades.F3 >= 1) {  
 if (leftGrades.F4 >= 1 && rightGrades.F4 >= 1) {  
 // F5 이상 체크...  
 return 'F5';  
 }  
 return 'F4';  
 }  
 return 'F3';  
 }  
 return 'F2';  
}

### 3.3 실제 트리 예시 (21노드)

1(F4)  
 / \  
 2(F3) 3(F3)  
 / \ / \  
 4(F2) 5(F2) 6(F2) 7(F3)  
 / \ / \ / \ / \  
 8(F1)... [리프 노드들] ...21(F1)

**등급 분포:** - F1: 11명 (리프) - F2: 6명 - F3: 3명 - F4: 1명 (루트)

## 4. 매출 및 지급 시스템

### 4.1 매출 및 지급 기준 (핵심 규칙 - v3.0 변경)

#### 4.1.1 매출 집계 기간 및 계산

**매출 집계 기간:**

매월 1일 00:00:00 ~ 말일 23:59:59

**기본 공식:**

월 총매출 = 해당 월 신규 가입 인원 × 100만원

**예시:**

9월 1일~30일 신규 가입: 10명  
9월 총매출: 1,000만원

#### 4.1.2 지급일 및 등급 기준 (중요 변경사항)

**지급일:**

매주 금요일

**등급 기준일 계산 공식:**

지급일: N년 M월 D일 (금요일)  
등급 기준일: N년 (M-1)월 (D-1)일 최종 등급  
  
예외 처리: 전월에 (D-1)일이 없는 경우 → 전월 말일 최종 등급

**구체적 예시:**

| 지급일 | 등급 기준일 계산 | 실제 등급 기준일 | 비고 |
| --- | --- | --- | --- |
| 10/3(금) | 9월 2일 | 9월 2일 23:59:59 | 정상 |
| 10/11(금) | 9월 10일 | 9월 10일 23:59:59 | 정상 |
| 11/1(금) | 10월 0일 | 9월 30일 23:59:59 | 10월 0일 없음 |
| 3/31(금) | 2월 30일 | 2월 28일 23:59:59 | 2월 30일 없음 |

**등급 기준일 계산 로직:**

function getGradeReferenceDate(paymentDate) {  
 const year = paymentDate.getFullYear();  
 const month = paymentDate.getMonth(); // 0-11  
 const day = paymentDate.getDate();  
   
 // 전월 (day-1)일 시도  
 let targetDate = new Date(year, month - 1, day - 1);  
   
 // 유효하지 않으면 전월 말일로  
 if (targetDate.getMonth() !== (month - 1 + 12) % 12) {  
 targetDate = new Date(year, month, 0); // 전월 말일  
 }  
   
 // 23:59:59 시점의 등급  
 targetDate.setHours(23, 59, 59, 999);  
   
 return targetDate;  
}

**매출 기준:**

지급일: N년 M월 D일  
매출 기준: N년 (M-1)월 1일~말일 총매출

#### 4.1.3 종합 지급 예시

**예시 1: 등급 유지**

지급일: 2024년 10월 3일 (금)  
등급 기준: 2024년 9월 2일 최종 등급 = F2  
매출 기준: 2024년 9월 총매출  
지급액: 9월 F2 지급액 100만원 ÷ 10 = 10만원

**예시 2: 등급 변동 (회차별 금액 변동)**

9월 등급 변화:  
- 9/1~9/6: F2  
- 9/7~9/30: F3  
  
10월 3일(금) 지급:  
- 등급 기준: 9/2 → F2  
- 지급액: F2 금액 10만원  
  
10월 11일(금) 지급:  
- 등급 기준: 9/10 → F3  
- 지급액: F3 금액 15만원  
  
→ 같은 9월 매출이지만 금요일마다 지급액 다름

**기존 규칙과의 차이:**

[기존 v2.0]  
- 매출 발생 월 등급 고정  
- 10회 모두 동일 금액  
  
[신규 v3.0]  
- 지급일 기준 전월 등급 사용  
- 회차마다 금액 다를 수 있음

### 4.2 등급별 지급 비율

| 등급 | 기본 비율 | 누적 계산식 |
| --- | --- | --- |
| F1 | 24% | (총매출×24%) ÷ (F1+F2인원) |
| F2 | 19% | F1지급액 + (총매출×19%) ÷ (F2+F3인원) |
| F3 | 14% | F2지급액 + (총매출×14%) ÷ (F3+F4인원) |
| F4 | 9% | F3지급액 + (총매출×9%) ÷ (F4+F5인원) |
| F5 | 5% | F4지급액 + (총매출×5%) ÷ (F5+F6인원) |
| F6 | 3% | F5지급액 + (총매출×3%) ÷ (F6+F7인원) |
| F7 | 2% | F6지급액 + (총매출×2%) ÷ (F7+F8인원) |
| F8 | 1% | F7지급액 + (총매출×1%) ÷ F8인원 |

**0명 처리 규칙:**

// 등급별 인원이 0명인 경우 처리  
if (gradeCount.F2 > 0 && gradeCount.F2 + gradeCount.F3 > 0) {  
 // 정상 계산  
 const f2Base = (totalRevenue \* 0.19) / (gradeCount.F2 + gradeCount.F3);  
 payments.F2 = Math.floor((f2Base + previousPayment) / 100) \* 100;  
} else {  
 // 0명인 경우 이전 등급 금액 유지  
 payments.F2 = previousPayment;  
}

**100원 단위 절삭:**

// Math.floor(금액 / 100) \* 100  
예: 1,234,567원 → 1,234,500원

**등급 고정 원칙 (중요):**

// 매출 발생 시점(N월)의 등급으로 지급액 확정  
// N+1월에 등급 변경되어도 N월 매출 지급액 불변  
// UserPaymentPlan.grade 필드에 고정값 저장  
  
예시:  
- 9월 매출 발생 시 홍길동 등급: F2  
- 10월에 F3으로 승급  
- 9월 매출 지급액은 여전히 F2 금액으로 10회 지급

### 4.3 10회 분할 지급 (v3.0 변경)

#### 4.3.1 지급 일정

**기본 원칙:** - **매주 금요일 지급** - 매출 발생 익월부터 **연속 10개 금요일**

**예시:**

9월 매출 발생  
  
10월 4일(금): 1회차 (등급 기준: 9/3)  
10월 11일(금): 2회차 (등급 기준: 9/10)  
10월 18일(금): 3회차 (등급 기준: 9/17)  
10월 25일(금): 4회차 (등급 기준: 9/24)  
11월 1일(금): 5회차 (등급 기준: 9/30, 10/0 없음)  
11월 8일(금): 6회차 (등급 기준: 10/7)  
11월 15일(금): 7회차 (등급 기준: 10/14)  
11월 22일(금): 8회차 (등급 기준: 10/21)  
11월 29일(금): 9회차 (등급 기준: 10/28)  
12월 6일(금): 10회차 (등급 기준: 11/5)

**금요일 스케줄 생성 로직:**

function generateFridaySchedule(revenueYear, revenueMonth) {  
 const schedules = [];  
 let currentDate = new Date(revenueYear, revenueMonth, 1); // 익월 1일  
   
 // 첫 금요일 찾기  
 while (currentDate.getDay() !== 5) { // 5 = 금요일  
 currentDate.setDate(currentDate.getDate() + 1);  
 }  
   
 // 연속 10개 금요일  
 for (let i = 1; i <= 10; i++) {  
 const gradeRefDate = getGradeReferenceDate(currentDate);  
   
 schedules.push({  
 installment: i,  
 paymentDate: new Date(currentDate),  
 gradeReferenceDate: gradeRefDate  
 });  
   
 currentDate.setDate(currentDate.getDate() + 7); // 다음 금요일  
 }  
   
 return schedules;  
}

#### 4.3.2 실제 계산 예시 (v3.0)

**시나리오:** - 9월 총매출: 1,000만원 - 김철수 9월 등급 변화: 9/1~9/10 F2, 9/11~9/30 F3 - 등급 분포: F1(50), F2(10), F3(4), F4(2)

**등급별 개인 지급액 계산:**

F2 개인 지급액:  
- F1 기본 = (1,000만원 × 24%) ÷ (50+10) = 40,000원  
- F2 추가 = (1,000만원 × 19%) ÷ (10+4) = 135,714원  
- F2 총액 = 40,000 + 135,714 = 175,714원 → 175,700원 (100원 절삭)  
  
F3 개인 지급액:  
- F2 누적 = 175,700원  
- F3 추가 = (1,000만원 × 14%) ÷ (4+2) = 233,333원  
- F3 총액 = 175,700 + 233,333 = 409,033원 → 409,000원 (100원 절삭)

**10회 지급 상세:**

10월 4일(금): 1회차  
- 등급 기준: 9/3 → F2  
- 지급: 175,700원 ÷ 10 = 17,570원  
  
10월 11일(금): 2회차  
- 등급 기준: 9/10 → F2  
- 지급: 175,700원 ÷ 10 = 17,570원  
  
10월 18일(금): 3회차  
- 등급 기준: 9/17 → F3  
- 지급: 409,000원 ÷ 10 = 40,900원  
  
... (이후 회차도 각 금요일의 등급 기준일에 따라 계산)

**핵심 차이점:** - v2.0: 10회 모두 동일 금액 - v3.0: 등급 변동 시 회차마다 금액 다름

### 4.4 중첩 지급 (v3.0 업데이트)

**여러 달 매출의 동시 지급:**

10월 4일(금) 수령 예시:  
  
김철수 등급 기준: 9월 3일 = F2  
  
├─ 7월 매출 7회차:  
│ 매출 기준: 7월  
│ 등급 기준: 9월 3일 F2  
│ 금액: 15,000원  
│  
├─ 8월 매출 3회차:  
│ 매출 기준: 8월  
│ 등급 기준: 9월 3일 F2  
│ 금액: 20,000원  
│  
└─ 9월 매출 1회차:  
 매출 기준: 9월  
 등급 기준: 9월 3일 F2  
 금액: 17,570원  
  
총 수령액: 52,570원  
원천징수(3.3%): 1,735원  
실지급액: 50,835원

**핵심 원칙:** - 같은 금요일에 지급되는 **모든 매출의 등급 기준일은 동일** - 10월 4일(금) → 모두 9월 3일 등급 기준 - 각 매출별 지급액은 **각 매출 발생 월의 총매출 기준으로 계산** - 등급만 동일 기준일 적용

**중첩 지급 계산 로직:**

async function calculateWeeklyPayment(userId, paymentDate) {  
 const gradeRefDate = getGradeReferenceDate(paymentDate);  
 const userGrade = await getUserGradeAt(userId, gradeRefDate);  
   
 let totalAmount = 0;  
 const details = [];  
   
 // 해당 금요일에 지급될 모든 매출 조회  
 const plans = await UserPaymentPlan.find({  
 userId: userId,  
 'installments.paymentDate': paymentDate  
 });  
   
 for (const plan of plans) {  
 // 각 매출별 등급별 지급액 계산 (동일 등급 기준일)  
 const monthlyRevenue = await MonthlyRevenue.findOne({  
 month: plan.revenueMonth  
 });  
   
 const gradePayment = calculateGradePayment(  
 monthlyRevenue.totalRevenue,  
 monthlyRevenue.gradeDistribution,  
 userGrade  
 );  
   
 const installmentAmount = Math.floor(gradePayment / 10 / 100) \* 100;  
 totalAmount += installmentAmount;  
   
 details.push({  
 revenueMonth: plan.revenueMonth,  
 installment: plan.currentInstallment,  
 amount: installmentAmount,  
 grade: userGrade  
 });  
 }  
   
 return {  
 totalAmount,  
 withholdingTax: Math.floor(totalAmount \* 0.033),  
 netAmount: Math.floor(totalAmount \* 0.967),  
 details  
 };  
}

### 4.5 원천징수

지급액: 40,905원  
원천징수(3.3%): 1,350원  
실지급액: 39,555원

### 4.6 등급별 지급금액 고정 설정 (v3.0 신규)

**기능:** - 관리자가 특정 매출월의 **등급별 지급금액을 일괄 지정** 가능 - 자동 계산 무시하고 **고정 금액으로 10회 분할 지급**

**데이터 모델:**

// MonthlyRevenue 확장  
{  
 month: String, // "2024-09"  
 totalRevenue: Number,  
 gradeDistribution: Object,  
   
 // 신규: 등급별 고정 금액  
 manualGradeAmounts: {  
 F1: Number, // null이면 자동 계산  
 F2: Number,  
 F3: Number,  
 F4: Number,  
 F5: Number,  
 F6: Number,  
 F7: Number,  
 F8: Number  
 }  
}  
  
// UserPaymentPlan 확장  
{  
 userId: String,  
 revenueMonth: String,  
 grade: String,  
 isManualOverride: Boolean, // 해당 등급이 수동 설정되었는지  
 totalAmount: Number, // 고정 또는 자동 계산 금액  
 // ... 기타 필드  
}

**적용 예시:**

2024년 9월 매출 등급별 고정 설정:  
  
관리자 설정:  
- F1: 50,000원  
- F2: 150,000원  
- F3: 350,000원  
- F4~F8: 미설정 (자동 계산)  
  
자동 계산 결과:  
- F2: 175,700원  
  
실제 지급:  
- 9월 F2 등급 모든 용역자: 150,000원 (고정)  
- 10월부터 10개 금요일: 회차당 15,000원  
- 등급 변동 무시  
- 원천징수 3.3% 적용

**등급별 고정 설정 처리:**

async function setMonthlyGradeAmounts(year, month, gradeAmounts) {  
 const monthKey = `${year}-${String(month).padStart(2, '0')}`;  
   
 // MonthlyRevenue에 등급별 고정 금액 저장  
 await MonthlyRevenue.findOneAndUpdate(  
 { month: monthKey },  
 {   
 manualGradeAmounts: {  
 F1: gradeAmounts.F1 || null,  
 F2: gradeAmounts.F2 || null,  
 F3: gradeAmounts.F3 || null,  
 F4: gradeAmounts.F4 || null,  
 F5: gradeAmounts.F5 || null,  
 F6: gradeAmounts.F6 || null,  
 F7: gradeAmounts.F7 || null,  
 F8: gradeAmounts.F8 || null  
 }  
 },  
 { upsert: true }  
 );  
   
 // 해당 월 모든 사용자 지급 계획 재생성  
 await regeneratePaymentPlans(year, month);  
}  
  
async function calculateUserPaymentAmount(userId, revenueMonth) {  
 const user = await User.findOne({ loginId: userId });  
 const monthlyRevenue = await MonthlyRevenue.findOne({   
 month: revenueMonth   
 });  
   
 // 등급별 고정 금액 확인  
 const manualAmount = monthlyRevenue.manualGradeAmounts?.[user.grade];  
   
 if (manualAmount) {  
 // 고정 금액 사용  
 return {  
 totalAmount: manualAmount,  
 isManualOverride: true,  
 installmentAmount: Math.floor(manualAmount / 10 / 100) \* 100  
 };  
 } else {  
 // 자동 계산 금액 사용  
 const autoAmount = calculateGradePayment(  
 monthlyRevenue.totalRevenue,  
 monthlyRevenue.gradeDistribution,  
 user.grade  
 );  
   
 return {  
 totalAmount: autoAmount,  
 isManualOverride: false,  
 installmentAmount: Math.floor(autoAmount / 10 / 100) \* 100  
 };  
 }  
}

## 5. 시스템 아키텍처

### 5.1 기술 스택

* **Backend**: Node.js, Express
* **Database**: MongoDB
* **Frontend**: SvelteKit
* **인증**: JWT + Session

### 5.2 핵심 서비스

#### 5.2.1 gradeCalculation.js (등급 계산 엔진)

**역할**: 이진 트리 기반 등급 계산

**주요 함수:** - calculateGradeForUser(userId): 개인 등급 계산 - 좌우 자식 확인 → F1/F2 판정 - 서브트리 등급 분포 수집 → F3~F8 조건 체크

* collectSubtreeGrades(userId): 서브트리 등급 분포 수집
  + DFS로 전체 하위 트리 순회
  + 등급별 카운트 반환: {F1: 5, F2: 1, ...}
* recalculateAllGrades(): 전체 사용자 등급 재계산
  + 리프 노드부터 시작하여 루트까지 상향식 계산

#### 5.2.2 batchProcessor.js (배치 프로세서)

**역할**: 용역자 등록 시 자동 배치 처리 엔진

**주요 함수:** - processNewUsers(userIds): 전체 자동 처리 1. recalculateGrades() - 전체 트리 등급 재계산 2. calculateMonthlyRevenue() - 월별 매출 계산 3. createPaymentSchedules() - 10주 지급 일정 생성 4. createUserPaymentPlans() - 개인별 지급 계획 생성

* calculateGradePayments(totalRevenue, gradeCount): 누적식 계산
  + F1부터 F8까지 순차적으로 계산
  + 이전 등급 금액 누적
  + 100원 단위 절삭
* generateInstallments(year, month): 10주 분할 스케줄 생성
  + N+1월부터 시작
  + 주차별 날짜 계산

#### 5.2.3 paymentService.js (지급 서비스)

**역할**: 주별 지급 실행 및 집계

**주요 기능:** - 특정 주차의 모든 지급 대상자 조회 - 다중 매출원의 회차 합산 (중첩 지급) - 원천징수 3.3% 계산 - WeeklyPayment 생성

#### 5.2.4 treeExtractor.js (트리 추출기)

**역할**: 트리 구조 최적화 추출

**주요 기능:** - 전체 트리 한 번 로드 - Map 구조로 메모리 계산 - 캐싱 (TTL: 1분)

#### 5.2.5 revenueService.js (매출 계산)

**역할**: 월별 매출 계산 및 관리

**주요 기능:** - calculateMonthlyRevenueForMonth(year, month): 특정 월 매출 계산 - 해당 월 신규 가입자 집계 (createdAt 기준) - 등급별 인원 분포 계산 - 등급별 지급액 계산 - MonthlyRevenue 저장

### 5.3 API 엔드포인트

| 엔드포인트 | 메소드 | 설명 |
| --- | --- | --- |
| /api/admin/users/bulk | POST | 엑셀 일괄 등록 |
| /api/admin/revenue/calculate | POST | 매출 계산 |
| /api/admin/payment/weekly | GET | 주간 지급 조회 |
| /api/tree/:userId | GET | 트리 구조 조회 |

### 5.4 성능 최적화

#### 5.4.1 인덱스 설정

// Users  
db.users.createIndex({ loginId: 1 }, { unique: true });  
db.users.createIndex({ parentId: 1 });  
db.users.createIndex({ grade: 1 });  
  
// PaymentPlans  
db.userpaymentplans.createIndex({  
 userId: 1,  
 revenueMonth: 1  
});

#### 5.4.2 배치 처리

* 개별 조회 대신 벌크 연산
* 100명 처리: 200회 → 2회 DB 접근
* 100배 성능 개선

## 6. 운영 지침

### 6.1 월간 운영 프로세스

[매월 말]  
1. 신규 등록 마감  
2. 총매출 집계  
3. 등급별 분포 확인  
4. 지급 계획 생성  
  
[다음달부터]  
5. 매주 자동 지급  
6. 지급 상태 모니터링

### 6.2 데이터 정합성

**검증 포인트:** - [ ] 총 지급액 = 총 매출액 - [ ] 10회 합계 = 개인별 총액 - [ ] 등급별 인원 × 개인액 = 등급 총액 - [ ] 이진트리 구조 무결성

### 6.3 백업 정책

* **일일 백업**: 전체 DB 스냅샷
* **주간 백업**: 지급 내역 아카이브
* **월간 백업**: 매출/등급 이력

### 6.4 보안 고려사항

* 관리자/용역자 권한 분리
* 지급 내역 변경 불가
* 모든 거래 로그 기록
* 개인정보 암호화

### 6.5 관리자 수동 작업

#### 6.5.1 F3 이상 보험 유지 조건 설정

**API 엔드포인트:** /api/admin/users/edit

**기능:** - 용역자 편집을 통해 보험 유지 조건 설정 - F3 이상 등급 용역자에 대해 적용

**데이터 모델 (User):**

{  
 // 기존 필드...  
 insuranceActive: Boolean, // 보험 유지 여부  
 insuranceAmount: Number, // 보험 유지 금액  
}

**처리 로직:**

async function updateUserInsurance(userId, insuranceData) {  
 const user = await User.findOne({ loginId: userId });  
   
 // F3 이상만 설정 가능  
 if (!['F3', 'F4', 'F5', 'F6', 'F7', 'F8'].includes(user.grade)) {  
 throw new Error('F3 이상 등급만 보험 조건 설정 가능');  
 }  
   
 await User.findOneAndUpdate(  
 { loginId: userId },  
 {  
 insuranceActive: insuranceData.active,  
 insuranceAmount: insuranceData.amount  
 }  
 );  
}

**지급 자격 검증 시 활용:**

// 매출 계산 시 F3 이상 보험 조건 체크  
if (['F3', 'F4', 'F5', 'F6', 'F7', 'F8'].includes(user.grade)) {  
 const requiredInsurance = getRequiredInsurance(user.grade);  
   
 if (!user.insuranceActive || user.insuranceAmount < requiredInsurance) {  
 // 지급 대상 제외  
 excludedUsers.push({  
 userId: user.loginId,  
 reason: 'INSURANCE\_NOT\_MET'  
 });  
 }  
}

## 부록

### A. 용어 정의

| 용어 | 설명 |
| --- | --- |
| **용역자** | MLM 참여자 |
| **판매인** | 신규 용역자의 추천인 |
| **이진트리** | 각 노드가 최대 2개 자식을 갖는 구조 |
| **누적식 배분** | 하위 등급 지급액을 상위가 포함 |
| **회차** | 10회 분할 지급의 각 단위 |

### B. 주요 명령어

# 개발 서버 실행  
pnpm run dev  
  
# 관리자 생성  
pnpm run create:admin  
  
# 데이터 초기화  
node scripts/reset-all-data.js  
  
# 매출 재계산  
node scripts/fix-monthly-revenue.js  
  
# 트리 구조 확인  
node scripts/check-tree-structure.js

### C. 트러블슈팅

| 문제 | 원인 | 해결 |
| --- | --- | --- |
| 매출 0원 | MonthlyRevenue import 누락 | 모델 import 확인 |
| 등급 오류 | 0명 처리 누락 | 분모 0 체크 추가 |
| 트리 깨짐 | 2단계 처리 오류 | 부모 존재 확인 |

**문서 끝**

*이 문서는 NanumPay MLM 시스템의 공식 설계 문서입니다.* *수정 및 배포 시 버전 관리를 준수하시기 바랍니다.* ### D. 핵심 파일 위치

#### 모델 (apps/web/src/lib/server/models/)

* User.js: 용역자 모델 (이진 트리 구조)
* MonthlyRevenue.js: 월별 매출
* UserPaymentPlan.js: 개인별 지급 계획
* WeeklyPayment.js: 주별 지급 실행

#### 서비스 (apps/web/src/lib/server/services/)

* gradeCalculation.js: 등급 계산 엔진
* batchProcessor.js: 자동 배치 처리
* paymentService.js: 지급 서비스
* revenueService.js: 매출 계산
* treeExtractor.js: 트리 구조 추출

#### API 엔드포인트 (apps/web/src/routes/api/admin/)

* users/bulk/+server.js: 엑셀 일괄 등록
* users/register/+server.js: 개별 등록
* revenue/calculate/+server.js: 매출 재계산
* payment/weekly/+server.js: 주별 지급

### E. 구현 상태 요약

#### ✅ 완전 구현

* 이진 트리 구조 (parentId, leftChildId, rightChildId)
* F1~F8 등급 자동 계산
* 누적식 지급액 계산
* 10회 분할 지급 스케줄
* 2단계 엑셀 등록 프로세스
* MLM 검증 룰 (루트 중복 방지, 자기 참조 방지)
* L→R 우선 배치
* 판매인 검색 우선순위
* 매출 귀속 월 자동 결정 (createdAt)
* 월별 그룹화 매출 계산
* 등급 고정 원칙
* 원천징수 3.3%
* 배치 처리 자동 트리거

#### ⚠️ 개선 필요

* 0명 처리: 현재 payments = 0, 문서 룰은 “이전 금액 유지”
* 순환 참조 방지: 자기 참조만 체크, A→B→C→A 미체크
* 트랜잭션 처리: 각 단계별 독립 실행
* 동시성 제어: Queue 또는 Lock 미구현

## 7. 등급별 최대 지급회수 제한 (신규 추가)

### 7.1 기본 원칙

**핵심 개념**: 같은 등급이 유지되면 최대 지급 회수 제한

| 등급 | 최대 지급 총회수 | 지급 가능 매출 기간 |
| --- | --- | --- |
| F1 | 20회 | 2개월 (20회 ÷ 10회/월) |
| F2 | 30회 | 3개월 |
| F3 | 40회 | 4개월 |
| F4 | 40회 | 4개월 |
| F5 | 50회 | 5개월 |
| F6 | 50회 | 5개월 |
| F7 | 60회 | 6개월 |
| F8 | 60회 | 6개월 |

**예시:** - F1 등급 유지 → 1월 매출 + 2월 매출 = 20회 지급 후 제외 - F2 등급 유지 → 1월 + 2월 + 3월 매출 = 30회 지급 후 제외

### 7.2 등급 상승 시 리셋

**원칙:** - 등급 상승하면 **지급 총회수 카운터 리셋** - 새 등급 기준으로 최대 회수 재적용

**예시:**

1월: F1 등급, 매출 발생 → 2월부터 10회 지급 (누적 10회)  
2월: F1 등급, 매출 발생 → 3월부터 10회 지급 (누적 20회) ← F1 최대치 도달  
3월: F2로 승급 → 카운터 0으로 리셋  
3월: F2 등급, 매출 발생 → 4월부터 10회 지급 가능 (누적 10회, 새 카운터)

### 7.3 F3 이상 추가 조건 (보험 유지금액)

**필수 보험 유지금액:** | 등급 | 최소 보험금액 | |——|————–| | F3, F4 | 50,000원 | | F5, F6 | 70,000원 | | F7, F8 | 100,000원 |

**조건 미달 시:** - 해당 월 매출의 **지급 대상에서 제외** - 등급은 유지되지만 지급만 중단

**보험 재가입 시:**

// 보험 해지: 지급 중단, 회수 카운터는 유지  
// 보험 재가입: 지급 재개, 기존 회수부터 계속  
  
예시:  
- 1월: F3, 보험 50,000원 → 10회 지급 (누적 10회)  
- 2월: F3, 보험 해지 → 지급 제외 (누적 10회 유지)  
- 3월: F3, 보험 50,000원 재가입 → 10회 지급 (누적 20회)

### 7.4 지급 자격 검증 로직

// batchProcessor.js - 매출 계산 시 검증  
async function calculateMonthlyRevenue(year, month) {  
 const newUsers = await User.find({  
 createdAt: {  
 $gte: new Date(year, month - 1, 1),  
 $lt: new Date(year, month, 1)  
 }  
 });  
  
 const eligibleUsers = [];  
 const excludedUsers = [];  
  
 for (const user of newUsers) {  
 // 1. 최대 회수 체크  
 const maxPayments = getMaxPaymentsByGrade(user.grade);  
 if (user.gradePaymentCount >= maxPayments) {  
 excludedUsers.push({  
 userId: user.loginId,  
 reason: 'MAX\_PAYMENT\_REACHED',  
 currentCount: user.gradePaymentCount,  
 maxCount: maxPayments  
 });  
 continue;  
 }  
  
 // 2. F3 이상 보험 조건 체크  
 if (['F3', 'F4', 'F5', 'F6', 'F7', 'F8'].includes(user.grade)) {  
 const requiredInsurance = getRequiredInsurance(user.grade);  
 if (!user.insuranceActive || user.insuranceAmount < requiredInsurance) {  
 excludedUsers.push({  
 userId: user.loginId,  
 reason: 'INSURANCE\_NOT\_MET',  
 currentInsurance: user.insuranceAmount || 0,  
 requiredInsurance  
 });  
 continue;  
 }  
 }  
  
 eligibleUsers.push(user);  
 }  
  
 // 적격자만 매출 계산  
 const totalRevenue = eligibleUsers.length \* 1000000;  
   
 return {  
 totalRevenue,  
 eligibleUsers,  
 excludedUsers  
 };  
}  
  
function getMaxPaymentsByGrade(grade) {  
 const limits = {  
 F1: 20, F2: 30, F3: 40, F4: 40,  
 F5: 50, F6: 50, F7: 60, F8: 60  
 };  
 return limits[grade] || 20;  
}  
  
function getRequiredInsurance(grade) {  
 const requirements = {  
 F3: 50000, F4: 50000,  
 F5: 70000, F6: 70000,  
 F7: 100000, F8: 100000  
 };  
 return requirements[grade] || 0;  
}

### 7.5 등급 변경 시 처리

// gradeCalculation.js - 등급 업데이트 시  
async function updateUserGrade(userId, newGrade) {  
 const user = await User.findOne({ loginId: userId });  
   
 if (user.grade !== newGrade) {  
 // 등급 상승 → 카운터 리셋  
 await User.findOneAndUpdate(  
 { loginId: userId },  
 {  
 grade: newGrade,  
 gradePaymentCount: 0, // 리셋  
 lastGradeChangeDate: new Date()  
 }  
 );  
   
 console.log(`등급 변경: ${user.name} ${user.grade} → ${newGrade}, 카운터 리셋`);  
 }  
}

### 7.6 지급 실행 시 카운터 증가

// paymentService.js - 주별 지급 실행  
async function executeWeeklyPayment(year, month, week) {  
 const payments = await UserPaymentPlan.find({  
 'installments.scheduledDate': { year, month, week },  
 isEligible: true // 적격자만  
 });  
  
 for (const plan of payments) {  
 // 지급 처리...  
 await createWeeklyPayment(plan);  
   
 // 사용자 카운터 증가  
 await User.findOneAndUpdate(  
 { loginId: plan.userId },  
 { $inc: { gradePaymentCount: 1 } }  
 );  
 }  
}

### 7.7 필요한 데이터 모델 확장

// User 모델 확장  
{  
 // 기존 필드...  
 grade: String,  
   
 // 신규 필드  
 gradePaymentCount: {  
 type: Number,  
 default: 0  
 },  
 lastGradeChangeDate: {  
 type: Date,  
 default: Date.now  
 },  
 insuranceAmount: {  
 type: Number,  
 default: 0  
 },  
 insuranceActive: {  
 type: Boolean,  
 default: false  
 }  
}  
  
// MonthlyRevenue 모델 확장  
{  
 // 기존 필드...  
   
 // 신규: 지급 제외자 목록  
 excludedUsers: [{  
 userId: String,  
 reason: String, // "MAX\_PAYMENT\_REACHED" | "INSURANCE\_NOT\_MET"  
 details: Object  
 }]  
}  
  
// UserPaymentPlan 모델 확장  
{  
 // 기존 필드...  
   
 // 신규: 지급 자격  
 isEligible: {  
 type: Boolean,  
 default: true  
 },  
 ineligibleReason: String,  
 gradePaymentCountAtCreation: Number // 생성 시점 카운터  
}

### 7.8 ⚠️ 기존 룰과의 충돌 해결 필요

**충돌 1: 등급 고정 vs 카운터 리셋**

문제:  
- 기존: "9월 F2 매출은 10월 F3 승급해도 F2 금액으로 10회 지급"  
- 신규: "등급 상승 시 카운터 리셋"  
  
해결 방안 (명확화 필요):  
A안) 카운터는 리셋되지만, 기존 매출 지급은 계속  
 - 9월 F2 매출: F2 금액으로 10회 지급 (카운터에 반영)  
 - 10월 F3 승급: 카운터 0으로 리셋  
 - 10월 F3 매출: 새로운 F3 금액으로 10회 지급 시작  
  
B안) 카운터 리셋 시 기존 지급도 중단  
 - 9월 F2 매출: 5회 지급 후 F3 승급 → 나머지 5회 중단

**충돌 2: 지급 회수 기준 모호**

질문: "지급 총회수"는 언제 카운트?  
- 매출 발생 시점? (9월 매출 발생 → 즉시 +10회)  
- 실제 지급 시점? (10월 1주차 지급 → +1회)  
  
권장: 실제 지급 시점 카운트

## 8. 구현 로드맵 (등급별 최대 지급회수 제한)

### Phase 1: 데이터 모델 확장

* ☐ User 모델에 gradePaymentCount, insuranceAmount, insuranceActive 추가
* ☐ MonthlyRevenue 모델에 excludedUsers 추가
* ☐ UserPaymentPlan 모델에 isEligible, ineligibleReason 추가

### Phase 2: 코어 로직 구현

* ☐ getMaxPaymentsByGrade() 함수 구현
* ☐ getRequiredInsurance() 함수 구현
* ☐ 매출 계산 시 지급 자격 검증 로직 추가
* ☐ 등급 변경 시 카운터 리셋 로직 추가
* ☐ 지급 실행 시 카운터 증가 로직 추가

### Phase 3: UI 구현

* ☐ 관리자 - 용역자 편집: 보험 금액/상태 입력 필드
* ☐ 관리자 - 용역자 목록: 지급 회수 표시
* ☐ 용역자 대시보드: 지급 회수 프로그레스바
* ☐ 용역자 대시보드: 보험 조건 표시

### Phase 4: 테스트 및 검증

* ☐ 최대 회수 도달 시나리오 테스트
* ☐ 등급 상승 시 카운터 리셋 테스트
* ☐ 보험 조건 미달 시나리오 테스트
* ☐ 보험 재가입 시 회수 유지 테스트

### Phase 5: 문서화

* ☐ 시스템\_설계\_문서.md 업데이트 (완료)
* ☐ API 문서 업데이트
* ☐ 운영 매뉴얼 업데이트