LFIT Admin API 아키텍처 문서

1. 프로젝트 구조

```
src/
├── application/        # 애플리케이션 서비스 계층
│   ├── ad.rewards.service.ts
│   ├── mission.rewards.service.ts
│   ├── ad.claims.service.ts
│   ├── mission.claims.service.ts
│   ├── jwt.service.ts
│   └── auth.service.ts
├── domain/            # 도메인 계층
│   ├── auth.ts
│   └── errors.ts
├── infrastructure/    # 인프라스트럭처 계층
│   ├── ad.rewards.repository.ts
│   ├── mission.rewards.repository.ts
│   ├── ad.claims.repository.ts
│   ├── mission.claims.repository.ts
│   └── dashboard.repository.ts
├── interfaces/        # 인터페이스 계층
│   ├── ad.rewards.controller.ts
│   ├── mission.rewards.controller.ts
│   ├── ad.claims.controller.ts
│   ├── mission.claims.controller.ts
│   ├── ad.rewards.routes.ts
│   ├── mission.rewards.routes.ts
│   ├── ad.claims.routes.ts
│   ├── mission.claims.routes.ts
│   └── auth.routes.ts
├── middleware/        # 미들웨어
│   └── auth.ts
├── types/            # 타입 정의
│   ├── ad.reward.ts
│   ├── mission.reward.ts
│   ├── ad.claim.ts
│   ├── mission.claim.ts
│   └── user.ts
└── app.ts            # 애플리케이션 진입점
```

2. 아키텍처 다이어그램

2.1 계층 구조

2.2 요청 흐름도

3. 계층별 상세 설명

3.1 인터페이스 계층 (interfaces/)

- HTTP 요청/응답 처리
- 입력 유효성 검증

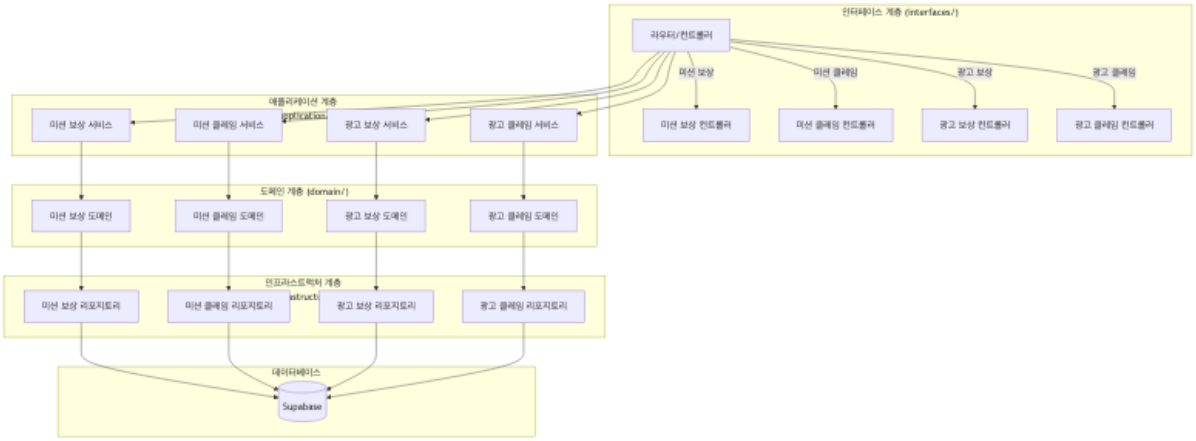

Figure 1: Architecture Diagram

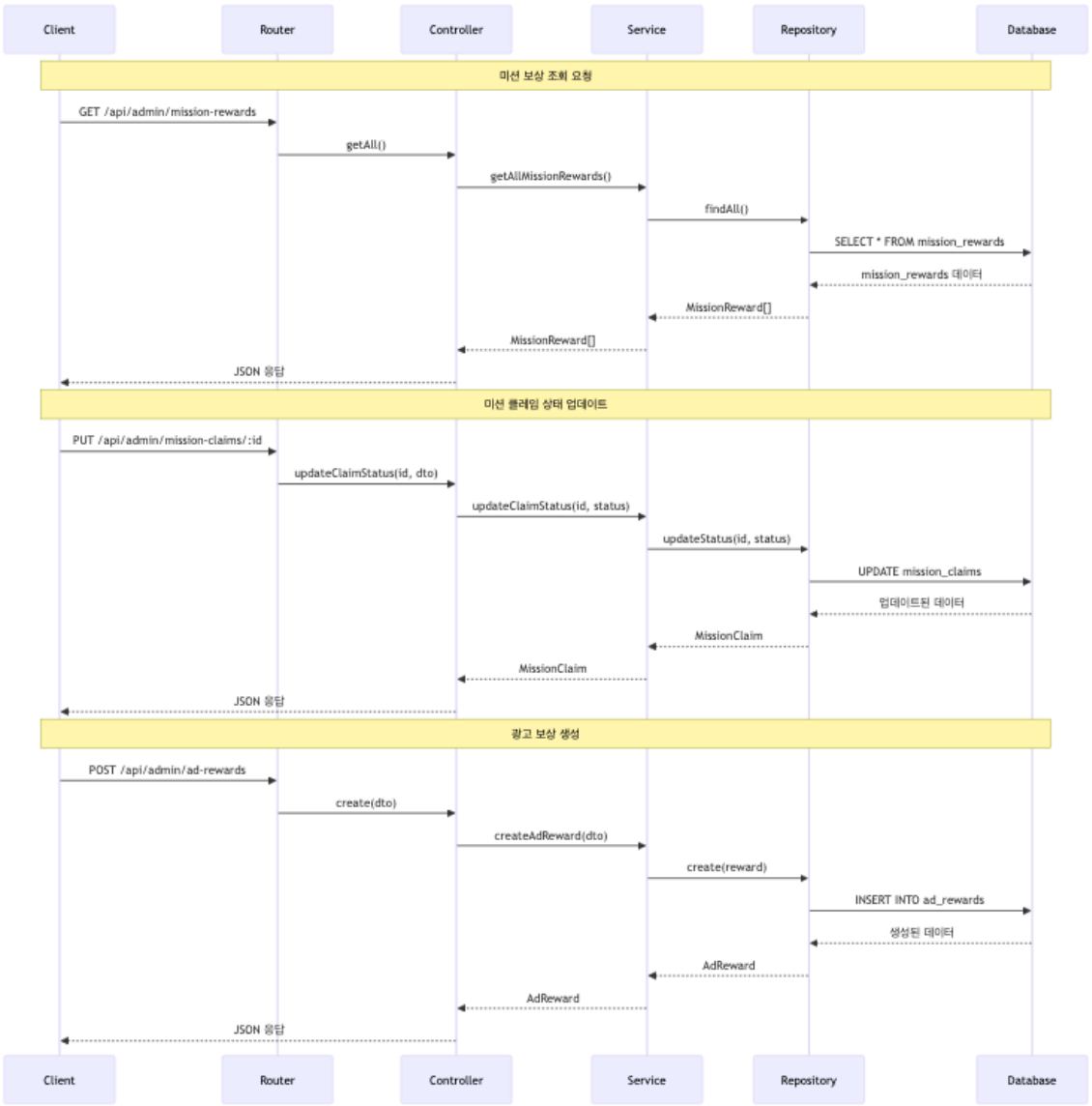

Figure 2: Flow Diagram

- 라우팅 설정
- 컨트롤러를 통한 요청 처리

컨트롤러 구조

```
// 미션 보상 컨트롤러 예시
export class MissionRewardsController {
    constructor(private readonly missionRewardsService: MissionRewardsService) {

    // CRUD 작업 메서드
    async getAll(): Promise<void>
    async getById(id: string): Promise<void>
    async create(dto: CreateMissionRewardDto): Promise<void>
    async update(id: string, dto: UpdateMissionRewardDto): Promise<void>
}

// 미션 클레임 컨트롤러 예시
export class MissionClaimController {
    constructor(private readonly missionClaimService: MissionClaimService) {}

    // 클레임 관련 메서드
    async getClaimsHistory(): Promise<void>
    async getClaimById(id: string): Promise<void>
    async updateClaimStatus(id: string, dto: UpdateClaimStatusDto): Promise<void
    async getClaimsStatistics(): Promise<void>
}
```

3.2 애플리케이션 계층 (application/)

- 비즈니스 로직 구현
- 트랜잭션 관리
- 도메인 객체 조작
- 서비스 간 조율

서비스 구조

```
// 미션 보상 서비스 예시
export class MissionRewardsService {
    constructor(private readonly repository: MissionRewardRepository) {}

    // 비즈니스 로직 메서드
    async getAllMissionRewards(): Promise<MissionReward[]>
    async createMissionReward(dto: CreateMissionRewardDto): Promise<MissionRewar
    async updateMissionReward(id: string, dto: UpdateMissionRewardDto): Promise<
}

// 미션 클레임 서비스 예시
export class MissionClaimService {
    constructor(private readonly repository: MissionClaimRepository) {}

    // 클레임 관련 메서드
    async getClaimsByDateRange(startDate: string, endDate: string): Promise<Miss
```

```
    async updateClaimStatus(id: string, status: MissionRewardClaimStatus): Promi
    async getClaimsStatistics(startDate: string, endDate: string): Promise<Missi
}
```

3.3 도메인 계층 (domain/)

- 핵심 비즈니스 규칙
- 도메인 모델 정의
- 도메인 이벤트

도메인 모델 예시

```
// 미션 보상 도메인 모델
export class MissionReward {
    private readonly id: string;
    private title: string;
    private reward: number;
    private isActive: boolean;

    constructor(props: MissionRewardProps) {
        this.validateProps(props);
        // ... 초기화 로직
    }

    // 도메인 규칙 및 비즈니스 로직
    activate(): void
    deactivate(): void
    updateReward(amount: number): void
}

// 미션 클레임 도메인 모델
export class MissionClaim {
    private readonly id: string;
    private userId: string;
    private missionId: string;
    private status: MissionRewardClaimStatus;
    private transactionHash?: string;

    constructor(props: MissionClaimProps) {
        this.validateProps(props);
        // ... 초기화 로직
    }

    // 도메인 규칙 및 비즈니스 로직
    updateStatus(status: MissionRewardClaimStatus): void
    setTransactionHash(hash: string): void
}
```

3.4 인프라스트럭처 계층 (infrastructure/)

- 데이터베이스 연동
- 외부 서비스 통합

- 저장소 구현

리포지토리 구조

```
// 미션 보상 리포지토리
export class MissionRewardRepository {
    constructor(private readonly db: Database) {}

    // CRUD 작업
    async findAll(): Promise<MissionReward[]>
    async findById(id: string): Promise<MissionReward>
    async create(reward: MissionReward): Promise<MissionReward>
    async update(id: string, reward: MissionReward): Promise<MissionReward>
}

// 미션 클레임 리포지토리
export class MissionClaimRepository {
    constructor(private readonly db: Database) {}

    // 클레임 관련 작업
    async findByDateRange(startDate: string, endDate: string): Promise<MissionCl
    async findById(id: string): Promise<MissionClaim>
    async updateStatus(id: string, status: MissionRewardClaimStatus): Promise<Mi
    async getStatistics(startDate: string, endDate: string): Promise<MissionClai
}
```

4. 데이터베이스 스키마

4.1 미션 보상 관련 테이블

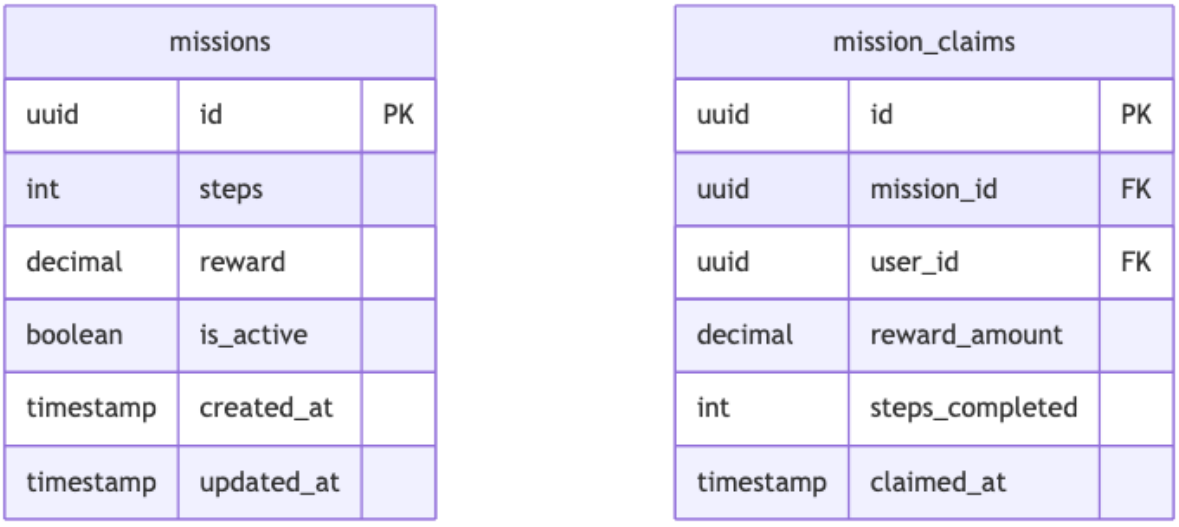

Figure 3: Mission Schema Diagram

ads		
uuid	id	PK
string	title	
decimal	reward	
boolean	is_active	
timestamp	created_at	
timestamp	updated_at	

ad_claims		
uuid	id	PK
uuid	ad_id	FK
uuid	user_id	FK
decimal	reward_amount	
timestamp	claimed_at	

Figure 4: Ad Schema Diagram

4.2 광고 보상 관련 테이블

5. API 상세 명세

5.1 미션 보상 API

엔드포인트	메서드	설명	권한
/api/admin/mission-rewards	GET	모든 미션 보상 조회	Admin
/api/admin/mission-rewards/:id	GET	특정 미션 보상 조회	Admin
/api/admin/mission-rewards	POST	새로운 미션 보상 생성	Admin
/api/admin/mission-rewards/:id	PUT	미션 보상 수정	Admin

5.2 미션 클레임 API

엔드포인트	메서드	설명	권한
/api/admin/mission-claims	GET	미션 클레임 내역 조회	Admin
/api/admin/mission-claims/:id	GET	특정 미션 클레임 조회	Admin
/api/admin/mission-claims/:id	PUT	미션 클레임 상태 업데이트	Admin
/api/admin/mission-claims/statistics	GET	미션 클레임 통계 조회	Admin

요청/응답 예시:

```
// POST /api/admin/mission-rewards
Request:
{
    "title": " 일일 로그인",
    "reward": 100,
```

```
    "daily_limit": 1
}

Response:
{
    "success": true,
    "data": {
        "id": "uuid",
        "title": " 일일 로그인",
        "reward": 100,
        "daily_limit": 1,
        "is_active": true,
        "created_at": "2024-03-20T00:00:00Z"
    }
}

// PUT /api/admin/mission-claims/:id
Request:
{
    "status": "COMPLETED",
    "transaction_hash": "0x123..."
}

Response:
{
    "success": true,
    "data": {
        "id": "uuid",
        "user_id": "user-uuid",
        "mission_id": "mission-uuid",
        "status": "COMPLETED",
        "transaction_hash": "0x123...",
        "updated_at": "2024-03-20T00:00:00Z"
    }
}
```

6. 보안 아키텍처

6.1 인증 흐름도

6.2 권한 체계

- USER: 일반 사용자 권한
- ADMIN: 관리자 권한
- SUPER_ADMIN: 최고 관리자 권한

7. 개선 필요 사항

7.1 도메인 계층 강화

- 도메인 이벤트 추가 필요
- 도메인 객체의 불변성 보장 강화

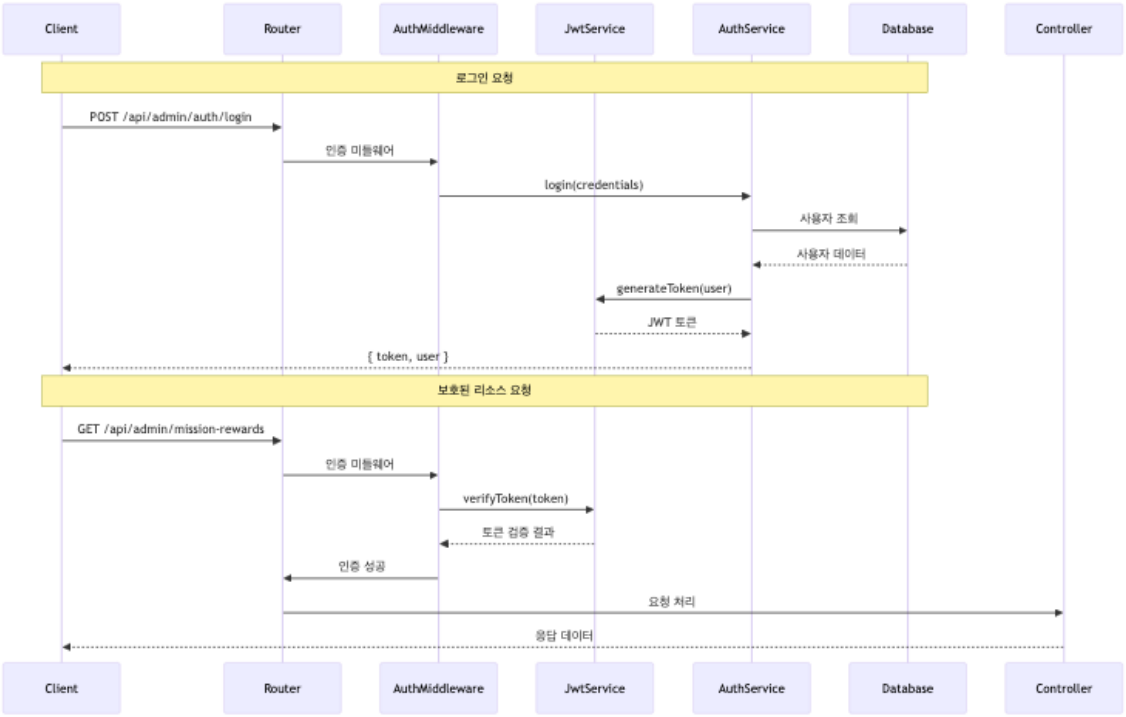

Figure 5: Auth Flow Diagram

- Value Object 패턴 적용 검토

7.2 보안 강화

- Rate Limiting 구현
- API 키 관리 체계 구축
- CORS 설정 상세화
- 입력값 검증 강화

7.3 에러 처리

```
// 글로벌 에러 핸들러 예시
app.use((err: Error, req: Request, res: Response, next: NextFunction) => {
    if (err instanceof ValidationError) {
        return res.status(400).json({
            success: false,
            error: {
                code: 'VALIDATION_ERROR',
                message: err.message,
                details: err.details
            }
        });
    }
    // ... 다른 에러 처리
});
```

7.4 성능 최적화

- 캐싱 전략 수립
  - Redis 를 활용한 캐시 레이어 구현
  - 캐시 무효화 전략 수립
- 데이터베이스 쿼리 최적화
  - 인덱스 최적화
  - 쿼리 실행 계획 분석
- N+1 문제 해결
  - DataLoader 패턴 적용
  - Join 쿼리 최적화

7.5 모니터링

- 로깅 시스템 구축
  - ELK 스택 도입
  - 로그 레벨 체계화
- 성능 메트릭 수집
  - Prometheus + Grafana 구축
  - 주요 지표 대시보드 구성
- 알림 시스템 구축
  - 임계치 기반 알림
  - 에러 발생 시 알림

8. 향후 개발 계획

8.1 단기 목표 (1-3 개월)

- 테스트 커버리지 향상
  - 단위 테스트 80% 이상
  - 통합 테스트 시나리오 구축
- API 문서화 (Swagger/OpenAPI)
  - 전체 API 엔드포인트 문서화
  - 예제 코드 추가
- 환경 설정 관리 개선
  - 환경변수 검증 로직 추가
  - 설정 값 중앙화

8.2 중기 목표 (3-6 개월)

- 마이크로서비스 아키텍처 검토
  - 서비스 분리 계획 수립
  - 서비스 간 통신 구조 설계
- 이벤트 기반 아키텍처 도입
  - 이벤트 버스 구축
  - 비동기 처리 로직 구현
- 실시간 처리 기능 강화
  - WebSocket 기반 실시간 알림
  - 실시간 데이터 동기화

8.3 장기 목표 (6 개월 이상)

- 확장 가능한 인프라 구축
  - 컨테이너 오케스트레이션
  - 자동 스케일링
- 데이터 분석 파이프라인 구축
  - 데이터 웨어하우스 구축
  - BI 도구 연동
- AI/ML 기능 통합
  - 사용자 행동 분석
  - 보상 최적화 알고리즘

9. 계층 간 관계 및 아키텍처 의의

9.1 계층 간 의존성 관계

9.1.1 도메인 계층과 인프라스트럭처 계층

```
// 도메인 계층은 인프라스트럭처 계층에 의존하지 않음
export interface IMissionRewardRepository {
    findAll(): Promise<MissionReward[]>;
    findById(id: string): Promise<MissionReward>;
    create(reward: MissionReward): Promise<MissionReward>;
    update(id: string, reward: MissionReward): Promise<MissionReward>;
}

// 인프라스트럭처 계층은 도메인 계층의 인터페이스를 구현
export class MissionRewardRepository implements IMissionRewardRepository {
    constructor(private readonly db: Database) {}
    // ... 구현
}
```

- **의존성 역전 원칙 (DIP) 적용**
  - 도메인 계층은 구체적인 구현체가 아닌 추상화된 인터페이스에 의존
  - 인프라스트럭처 계층이 도메인 계층의 인터페이스를 구현
  - 데이터베이스나 외부 서비스 변경 시 도메인 로직에 영향 없음

9.1.2 애플리케이션 계층과 인프라스트럭처 계층

```
// 애플리케이션 계층은 추상화된 인터페이스에 의존
export class MissionRewardsService {
    constructor(private readonly repository: IMissionRewardRepository) {}

    async createMissionReward(dto: CreateMissionRewardDto): Promise<MissionRewar
        const reward = new MissionReward(dto);
        return this.repository.create(reward);
    }
}
```

- **의존성 주입 (DI) 패턴 적용**
  - 서비스는 구체적인 리포지토리 구현체가 아닌 인터페이스에 의존
  - 테스트 시 모의 객체 (Mock) 주입 가능
  - 런타임에 실제 구현체 주입

9.1.3 인터페이스 계층과 애플리케이션 계층

```
// 컨트롤러는 서비스 인터페이스에 의존
export class MissionRewardsController {
    constructor(private readonly missionRewardsService: IMissionRewardsService)

    async create(req: Request, res: Response): Promise<void> {
        const dto = this.validateCreateDto(req.body);
        const result = await this.missionRewardsService.createMissionReward(dto)
        res.json({ success: true, data: result });
    }
}
```

- **단일 책임 원칙 (SRP) 적용**
  - 컨트롤러는 HTTP 요청/응답 처리에만 집중
  - 비즈니스 로직은 서비스 계층에 위임
  - 입력 유효성 검증과 응답 포맷팅에 집중

9.2 모던 아키텍처에서의 의의

9.2.1 관심사의 분리

- **계층별 독립성**
  - 각 계층이 자신의 책임에만 집중
  - 다른 계층의 변경에 영향을 최소화
  - 코드의 재사용성과 유지보수성 향상
- **테스트 용이성**
  - 각 계층을 독립적으로 테스트 가능
  - 모의 객체를 통한 격리된 테스트
  - 테스트 커버리지 향상 용이

9.2.2 확장성과 유연성

- **기술 스택 변경 용이**
  - 데이터베이스 변경 시 리포지토리 계층만 수정
  - 외부 서비스 변경 시 해당 어댑터만 수정
  - 도메인 로직은 변경 불필요
- **기능 확장 용이**
  - 새로운 기능 추가 시 기존 계층 구조 유지
  - 인터페이스를 통한 확장 가능
  - 기존 코드 수정 최소화

9.2.3 비즈니스 로직 보호

- **도메인 규칙 캡슐화**
  - 핵심 비즈니스 규칙을 도메인 계층에 격리
  - 외부 의존성으로부터 보호
  - 규칙 변경 시 영향 범위 최소화
- **일관성 유지**
  - 도메인 규칙이 한 곳에서 관리됨
  - 규칙 위반 가능성 감소
  - 비즈니스 정책 변경 용이

9.3 실제 적용 사례

9.3.1 미션 보상 시스템

```
// 도메인 계층
export class MissionReward {
    private readonly id: string;
    private title: string;
    private reward: number;
    private isActive: boolean;

    // 도메인 규칙
    activate(): void {
        if (this.reward <= 0) {
            throw new Error('보상 금액은 0 보다 커야 합니다');
        }
        this.isActive = true;
    }
}

// 인프라스트럭처 계층
export class MissionRewardRepository implements IMissionRewardRepository {
    constructor(private readonly db: Database) {}

    async create(reward: MissionReward): Promise<MissionReward> {
        const { data, error } = await this.db
            .from('mission_rewards')
            .insert(reward)
            .select()
            .single();

        if (error) throw new Error('미션 보상 생성 실패');
        return data;
    }
}

// 애플리케이션 계층
export class MissionRewardsService {
    constructor(private readonly repository: IMissionRewardRepository) {}

    async createMissionReward(dto: CreateMissionRewardDto): Promise<MissionRewar
        const reward = new MissionReward(dto);
        reward.activate(); // 도메인 규칙 적용
        return this.repository.create(reward);
    }
}
```

9.3.2 클레임 처리 시스템

```
// 도메인 계층
export class MissionClaim {
    private readonly id: string;
```

```
    private status: MissionRewardClaimStatus;
    private transactionHash?: string;

    // 도메인 규칙
    updateStatus(status: MissionRewardClaimStatus): void {
        if (this.status === 'COMPLETED') {
            throw new Error('이미 완료된 클레임입니다');
        }
        this.status = status;
    }
}

// 인프라스트럭처 계층
export class MissionClaimRepository implements IMissionClaimRepository {
    constructor(private readonly db: Database) {}

    async updateStatus(id: string, status: MissionRewardClaimStatus): Promise<Mi
        const { data, error } = await this.db
            .from('mission_claims')
            .update({ status })
            .eq('id', id)
            .select()
            .single();

        if (error) throw new Error('클레임 상태 업데이트 실패');
        return data;
    }
}

// 애플리케이션 계층
export class MissionClaimService {
    constructor(private readonly repository: IMissionClaimRepository) {}

    async updateClaimStatus(id: string, status: MissionRewardClaimStatus): Promi
        const claim = await this.repository.findById(id);
        claim.updateStatus(status); // 도메인 규칙 적용
        return this.repository.updateStatus(id, status);
    }
}
```

이러한 계층화된 아키텍처는 다음과 같은 이점을 제공합니다:

1. **유지보수성**
   - 각 계층의 책임이 명확히 구분됨
   - 코드 변경의 영향 범위가 제한적
   - 버그 수정과 기능 추가가 용이
2. **테스트 용이성**
   - 각 계층을 독립적으로 테스트 가능
   - 모의 객체를 통한 격리된 테스트
   - 테스트 커버리지 향상
3. **확장성**
   - 새로운 기능 추가가 용이

    - 기존 코드 수정 최소화
    - 기술 스택 변경 용이
4. **비즈니스 로직 보호**
    - 핵심 규칙이 도메인 계층에 격리
    - 외부 의존성으로부터 보호
    - 규칙 변경 시 영향 범위 최소화