ChatGPT

온라인 평가 시스템 코드 분석 및 문제 해결 계획

본 분석에서는 GitHub 저장소 **bruce0817kr/Online-evaluation**의 백엔드/프론트엔드 코드를 종합적으로 검토하고, 지적된 주요 문제들의 원인과 해결 방안을 제시합니다. 특히, **(1)** 백엔드 서버 실행 시 발생하는 ImportError, **(2)** 서버 코드에서 참조되었으나 정의되지 않은 모델/함수 문제, **(3)** 관리자·간사·평가위원 로그인 실패 문제, **(4)** 프론트엔드와 백엔드 API 연동 오류, **(5)** Docker를 활용한 서비스 실행 환경 점검 및 개선 방안에 대해 다룹니다. 각 문제에 대한 원인 분석, 코드 구조/관계 설명, 수정 계획, 필요 시 예시 코드를 포함하여 제시합니다.

시스템 구조 개요

본 프로젝트는 **FastAPI** 기반 백엔드와 **React** 기반 프론트엔드로 구성된 **온라인 평가 관리 시스템**입니다. 주요 구성 요소는 다음과 같습니다:

- **백엔드 (FastAPI)** – `backend/server.py`가 메인 어플리케이션으로, 인증(OAuth2 JWT), 사용자관리, 프로젝트/회사/평가 관리, 파일 업로드, 웹소켓 알림 등을 처리합니다. **MongoDB** (프로젝트/사용자/평가 데이터 저장)와 **Redis** (캐시/세션 관리)를 사용하며, Pydantic 모델(`backend/models.py`)로 데이터 스키마를 정의하고 보안 유틸(`backend/security.py`), 캐시 서비스(`backend/cache_service.py`), 헬스 모니터링(`backend/health_monitor.py`), WebSocket 서비스(`backend/websocket_service.py`) 등 모듈로 기능을 분리합니다 [1]. 백엔드는 Docker 컨테이너로 구동되며 Uvicorn/Gunicorn으로 서비스됩니다.

- **프론트엔드 (React)** – `frontend/` 디렉토리에 위치하며, Tailwind CSS를 사용한 UI와 Axios 기반 API 연동으로 구성됩니다. 로그인 화면, 평가 템플릿 관리(`TemplateManagement.js`), 평가 진행/조회(`EvaluationManagement.js`) 등의 컴포넌트가 있으며, `.env`를 통해 `REACT_APP_BACKEND_URL`(예: `http://localhost:8080/api`) 설정 후 백엔드와 통신합니다.

- **Docker 환경** – `docker-compose.yml`로 **MongoDB**, **Redis**, **백엔드**, **프론트엔드**, (선택적 **Nginx**) 서비스를 정의하고 있습니다 [2] [3]. 개발용/운영용 Dockerfile을 구분하여 멀티스테이지 빌드와 핫리로드(Uvicorn `--reload`) 등을 지원합니다 (`Dockerfile.backend`의 `development` 스테이지 등). 서비스 기동 시 헬스체크(`/health` 엔드포인트)와 기본 데이터베이스 연결 확인 등을 수행하며, 초기 관리자 계정 생성을 위한 `/api/init` 엔드포인트도 제공합니다 [4].

1. ImportError – 상대 경로 임포트 오류

문제 현상: 백엔드 실행 시 `backend/server.py`에서 `cache_service` 모듈을 임포트할 때 `ImportError: attempted relative import with no known parent package` 오류가 발생합니다. 이는 `server.py` 상단에서 다음과 같이 **상대 경로**로 임포트하고 있기 때문입니다:

```
# server.py (일부 발췌)
from .cache_service import cache_service   # Added import (상대경로 임포트) [5]
```

FastAPI 앱을 `uvicorn`으로 실행할 때 (`python -m uvicorn server:app` 명령) `server.py`를 최상위 모듈로 취급하므로, `from .cache_service`와 같은 상대 임포트는 **부모 패키지**를 알 수 없어 실패하게 됩니다

[5]. 즉, `backend` 디렉토리가 패키지로 인식되지 않았거나, `server.py`를 패키지 모듈로 실행하지 않아 발생하는 문제입니다.

원인 분석: 프로젝트 구조상 `backend/` 디렉토리에 `__init__.py` 파일이 없고, Docker 빌드 시 `backend/` 디렉토리 내용을 `/app`에 복사하여 모듈 경로가 평탄화(flatten)되고 있습니다. 그 결과 `server.py`와 `cache_service.py`가 동일 경로에 위치하지만, `server.py` 내의 `from .cache_service` 문장은 "현재 패키지에서 cache_service 모듈을 가져온다"는 의미로 해석되어 **오류**가 발생합니다. 패키지 컨텍스트가 없으므로 Python은 `.`(현재 패키지)을 해석하지 못합니다.

해결 방안: 모듈 임포트 경로를 **절대 경로**로 수정하거나 프로젝트 구조를 패키지화해야 합니다. 구체적으로:

- **절대 임포트로 수정:** `server.py`에서 `from .cache_service import cache_service`를 `from backend.cache_service import cache_service`처럼 **프로젝트 최상위 기준**으로 수정합니다. 이렇게 하면 `backend` 디렉토리를 패키지로 간주하지 않고도 임포트가 가능합니다. 또는, 아예 `cache_service.py`가 `server.py`와 같은 디렉토리에 있으므로 `import cache_service`로 변경하고 사용 시 `cache_service.cache_service`로 접근할 수도 있습니다. 중요한 것은 임포트 시 `.`와 같은 패키지 상대 표기를 피하는 것입니다.

- **패키지 구조 반영:** `backend/` 디렉토리에 `__init__.py`를 추가하여 패키지로 만든 후, `server.py`를 모듈로 실행할 때 `python -m backend.server` 형태로 실행하는 방법입니다. 이 경우 `from .cache_service import cache_service`도 정상 동작합니다. 하지만 Docker Compose 설정을 비롯한 실행 커맨드를 모두 `backend.server:app` 형태로 바꿔야 하므로, **보다 간단한 해결**은 위의 절대 임포트 방식입니다.

위 조치 후에는 ImportError 없이 `server.py`가 로드되어 **Redis 캐시 서비스**(`cache_service.redis_client` 등) 초기화 코드가 정상 수행될 것입니다[6]. 실제 프로젝트에서는 이러한 임포트 문제를 예방하기 위해 PYTHONPATH 환경변수에 `/app`을 추가하거나(이미 Dockerfile에서 `ENV PYTHONPATH=/app` 설정[7]) 절대경로 임포트를 사용하는 것이 권장됩니다.

2. 미정의 모델 및 함수 – Pydantic 모델/유틸리티 누락

문제 현상: `server.py`에서 사용되는 일부 **Pydantic 모델**과 **함수**들이 정의되어 있지 않아, 코드 린트 단계나 실행 시 참조 오류가 발생합니다. 구체적으로 언급된 항목은:

- **Pydantic 모델 누락:** `EvaluatorCreate`, `Project`, `Company`, `FileMetadata` 등이 임포트 또는 사용되지만 정의가 없음.
- **함수 누락:** `generate_evaluator_credentials`, `update_project_statistics` 등이 사용되나 구현되지 않음.

이들은 주로 **새 평가위원 생성**, **프로젝트/회사 생성 시 통계 업데이트**, **파일 메타데이터 관리** 등 핵심 로직에 해당합니다. 코드 일부를 보면, 예를 들어 평가위원 생성 시 아래와 같이 사용되고 있습니다:

```
# 평가위원 생성 API (일부 발췌)
@api_router.post("/evaluators")
async def create_evaluator(evaluator_data: EvaluatorCreate, ...):
    ...
    login_id, password = generate_evaluator_credentials(evaluator_data.user_name,
evaluator_data.phone)【43†L722-L730】
```

```
    ...
    user = User(
        login_id=login_id,
        password_hash=hashed_password,
        user_name=evaluator_data.user_name,
        email=evaluator_data.email,
        phone=evaluator_data.phone,
        role="evaluator"
    )
    ...
    return { **UserResponse(**user.dict()).dict(), "generated_login_id": login_id,
"generated_password": password, ... }
```

또한 프로젝트/회사 생성 시 통계 업데이트를 위해 배경 작업에 `update_project_statistics`를 추가하지만 실제 구현이 없습니다:

```
# 프로젝트 생성 API (일부 발췌)
project = Project(...); await db.projects.insert_one(project.dict())
background_tasks.add_task(update_project_statistics, project.id)【43†L836-L840】

# 회사 생성 API (일부 발췌)
company = Company(...); await db.companies.insert_one(company.dict())
background_tasks.add_task(update_project_statistics, company.project_id)【43†L862-L868】
```

그리고 파일 업로드 처리 시 파일 메타데이터 객체를 만들지만 `FileMetadata` 클래스 정의가 없어 오류가 발생합니다:

```
# 파일 업로드 API (일부 발췌)
file_metadata = FileMetadata(
    id=file_id,
    filename=unique_filename,
    original_filename=file.filename,
    file_path=str(file_path),
    file_size=len(content),
    file_type=file.content_type,
    uploaded_by=current_user.id,
    company_id=company_id
)【43†L896-L904】
await db.file_metadata.insert_one(file_metadata.dict())  # DB에 메타데이터 저장
```

원인 분석: 이러한 모델/함수는 기획 단계에서 필요성이 파악되어 코드에서 호출하고 있으나, 아직 `models.py` **등 해당 위치에 정의되지 않았거나** 부분적으로만 정의된 상태입니다. 실제 `backend/models.py`를 확인해보면 `EvaluatorCreate`, `Project`, `Company` 등 일부 모델은 정의되어 있으나[10][11][12], `FileMetadata` 모델이나 `EvaluationTemplateCreate` 모델 등은 누락되어 있습니다. 마찬가지로 `generate_evaluator_credentials`와 `update_project_statistics` 함수는 `security.py`나 `server.py` 어디에도 정의가 없어 NameError가 발생합니다. 이러한 누락은 이전 작업 요약에서도 "서버 코드 불완전 문제"로 지적되어 있으며, 여러 함수 구현이 빠져 있음이 언급되었습니다[13][14].

해결 방안: 누락된 모델과 함수를 **적절한 위치에 정의**하여 오류를 해소해야 합니다. 구체적인 보완 계획은 다음과 같습니다:

- **EvaluatorCreate 모델** – `평가위원 생성` 시 사용되는 입력 스키마로, 이미 `models.py`에 아래와 같이 정의되어 있습니다 [10]. 필요에 따라 필드를 추가합니다. 현재는 이름, 전화번호, 이메일 정도로 구성되어 있는데, 로그인 ID는 생성 시 자동 부여되므로 포함하지 않습니다.

```
class EvaluatorCreate(BaseModel):
    user_name: str
    phone: str
    email: EmailStr
    # 필요한 경우 다른 필드 추가 (예: 소속 등)
```

현 코드 상으로는 충분하며, 해당 모델이 `server.py`에서 `User` 생성에 활용됩니다.

- **Project / ProjectCreate 모델** – 프로젝트 객체의 생성/응답 모델로, `models.py`에 기본적인 필드가 정의되어 있습니다 [15]. 예를 들어 `Project`에는 `id, name, description, start_date, end_date, created_at, created_by` 등이 있습니다. 추가로 프로젝트 마감일이나 상태필드가 필요하다면 `ProjectBase`/`ProjectCreate`에 확장할 수 있습니다. 위 코드에서는 `deadline` 처리를 하고 있으므로, `ProjectCreate` 모델에 `deadline: datetime` 필드를 포함해야 합니다 (없을 경우 `deadline` 속성 접근 시 오류 발생). 또한 `Company.project_id` 필드도 코드에서 사용되므로, Company 모델 정의에 `project_id: str` 필드를 추가해야 합니다.

- **Company / CompanyCreate 모델** – 기업 객체의 생성/응답 모델로, `models.py`에 `name, registration_number, address` 등의 기본 필드가 있습니다 [16]. 코드에서는 회사 생성 시 `company.project_id`를 설정하므로, Company 모델에 `project_id: str` 필드를 추가 정의해야 합니다. 예:

```
class CompanyBase(BaseModel):
    name: str
    registration_number: Optional[str] = None
    address: Optional[str] = None
    project_id: str   # 프로젝트 ID 참조 필드 추가
```

위처럼 필드를 보완하고, `Company` 모델에도 반영합니다.

- **FileMetadata 모델** – 업로드된 파일의 메타정보를 저장하는 Pydantic 모델을 새로 정의해야 합니다. 이는 파일 ID, 이름, 원본 이름, 저장 경로, 크기, 타입, 업로더, 소속 회사 등을 포함합니다 [9]. `models.py`에 예를 들어 다음과 같이 추가합니다:

```
class FileMetadata(BaseModel):
    id: str = Field(default_factory=lambda: str(uuid.uuid4()), alias="_id")
    filename: str
    original_filename: str
    file_path: str
    file_size: int
    file_type: str
```

```
    uploaded_by: str
    company_id: str
    uploaded_at: datetime = Field(default_factory=datetime.utcnow)

    class Config:
        json_encoders = {datetime: lambda dt: dt.isoformat()}
```

설명: `id`는 고유 식별자(UUID)로 생성하며, MongoDB에 저장할 때 `_id`로 사용할 수 있도록 alias를 지정했습니다. `uploaded_at`을 기록하여 파일 업로드 시점을 추적할 수 있습니다. 이 모델 정의 후 `server.py`에서 `FileMetadata(...)`로 객체를 생성해도 오류 없이 동작하며, 이를 `db.file_metadata` 컬렉션에 저장할 수 있습니다.

- **generate_evaluator_credentials 함수** – 평가위원의 **로그인 ID와 초기 비밀번호를 자동 생성**하는 유틸 함수입니다. 코드상 의도는 평가위원 이름과 전화번호를 이용하여 고유한 자격증명을 만드는 것입니다 [8]. 예컨대 현재 구현 의도는 "이름을 login_id로, 전화번호를 숫자만 추출해 비밀번호로 사용"하는 것으로 보입니다 [17] [18]. 이를 구현하기 위해 `security.py` 또는 `server.py`에 다음과 같은 함수를 정의할 수 있습니다:

```
import re

def generate_evaluator_credentials(name: str, phone: str) -> (str, str):
  """평가위원 이름과 전화번호로 고유 아이디/비밀번호 생성"""
    # 로그인 ID 생성: 이름의 공백 제거 및 소문자화
    login_id = re.sub(r'\s+', '', name)
    login_id = login_id.lower()
    # 전화번호 숫자만 추출하여 비밀번호 사용 (예: "010-1234-5678" -> "01012345678")
    raw_phone = re.sub(r'\D', '', phone)
    password = raw_phone if raw_phone else "password123"
    return login_id, password
```

위 함수는 단순 예시입니다. 실제로는 **동명이인 충돌 방지**를 위해 이름 이외에 난수나 ID를 조합하거나, 보안 강화를 위해 비밀번호 생성 규칙을 강화하는 것이 좋습니다. 생성된 `login_id`와 `password`는 `create_evaluator` API 내부에서 사용되며, `password`는 곧바로 해싱하여 DB에 `password_hash`로 저장됩니다 [19]. 또한 API 응답으로 생성된 ID/비밀번호를 반환하여 관리자에게 안내하고 있습니다 [20].

- **update_project_statistics 함수** – 프로젝트별 평가 진행 통계를 갱신하는 백그라운드 작업 함수입니다. 프로젝트 생성, 회사 추가, 과제(assignment) 생성, 평가 제출 등 이벤트 발생 시 해당 프로젝트의 통계를 계산하여 DB에 저장하는 역할을 합니다 [21] [22]. 이 함수를 구현하지 않으면 이러한 이벤트에서 NameError가 발생하므로, `server.py` 하단이나 별도 모듈에 다음과 같이 작성할 수 있습니다:

```
async def update_project_statistics(project_id: str):
  """주어진 프로젝트의 통계를 집계하여 저장"""
    # 총 할당된 평가 수 (evaluation_sheets 컬렉션 기준)
    total_assigned = await db.evaluation_sheets.count_documents({"project_id": project_id})
    # 완료된 평가 수 (status가 submitted 또는 completed인 경우)
    completed = await db.evaluation_sheets.count_documents({
        "project_id": project_id, "status": {"$in": ["submitted", "completed"]}
    })
```

```
    # 평균 점수 계산 (완료된 평가에 한해)
    average_score = None
    if completed > 0:
        # evaluation_sheets에 total_score 필드가 누적된 총점이라고 가정
        result = await db.evaluation_sheets.aggregate([
            {"$match": {"project_id": project_id, "status": {"$in": ["submitted", "completed"]}}},
            {"$group": {"_id": None, "avg_score": {"$avg": "$total_score"}}}
        ]).to_list(1)
        if result:
            average_score = result[0].get("avg_score")
    # 통계 모델 작성 (ProjectStatistics 활용)
    stats = {
        "project_id": project_id,
        "total_assigned_evaluations": total_assigned,
        "completed_evaluations": completed,
        "average_score": average_score
    }
    # 통계 컬렉션에 업sert (기존 통계 문서가 있으면 갱신, 없으면 생성)
    await db.project_statistics.update_one(
        {"project_id": project_id}, {"$set": stats}, upsert=True
    )
```

위 코드는 **예시 구현**으로, 실제 필드명이나 컬렉션명은 프로젝트에 맞게 조정해야 합니다. `ProjectStatistics` Pydantic 모델이 `models.py`에 정의되어 있으므로 [23], 해당 구조에 따라 통계 정보를 저장/조회할 수 있습니다. 예컨대 별도 `project_statistics` 컬렉션을 사용하거나, 간단히 `projects` 컬렉션의 해당 프로젝트 도큐먼트에 통계 필드를 추가하는 방식 중 하나를 선택할 수 있습니다. 여기서는 별도 컬렉션을 가정했습니다. 이 함수가 구현되면, 프로젝트/회사 생성, 평가 할당, 평가 제출 등의 API에서 백그라운드로 호출되어 통계를 자동 갱신합니다.

- **기타 모델/함수** – 이 외에 코드 전반을 점검하여, 예컨대 `EvaluationTemplateCreate` 모델 (템플릿 생성시 사용) 등 누락된 부분도 함께 정의해야 합니다. 현재 `server.py`의 `/api/templates` POST 엔드포인트 시그니처를 보면 `template_data: EvaluationTemplateCreate`로 되어 있으므로 [24], 해당 모델도 `models.py`에 추가해야 합니다. Template 생성에 필요한 `name, description, items(List[EvaluationItem])` 등을 포함하는 모델로 작성하면 됩니다. 프론트엔드와의 연동을 고려하여, 하단 **프론트엔드 연동 오류** 섹션에서 추가로 언급하겠습니다.

이상 조치들을 통해, 린터 오류나 NameError 없이 **모든 참조가 실제 정의된 모델/함수를 가리키도록 수정**할 것입니다. 필요하다면 오늘 작업 요약서 등 내부 문서를 참고하여 누락된 부분을 하나씩 구현해야 합니다 [14]. 최종적으로 `server.py` 내 임포트 구문도 조정하여, 예를 들어 `from models import EvaluatorCreate, Project, ProjectCreate, Company, CompanyCreate, FileMetadata, ...` 등으로 새로운 클래스들을 임포트하도록 수정합니다.

3. 로그인 실패 및 API 연동 문제 – 인증 시스템 불안정

문제 현상: 관리자(Admin), 간사(Secretary), 평가위원(Evaluator) 계정으로 **로그인이 지속적으로 실패**합니다. 사용자가 올바른 자격증명을 입력해도 토큰이 발급되지 않거나 인증 오류(HTTP 401)가 발생한다고 보고되었습니다. 백엔드 API가 불안정하여 인증 관련 엔드포인트가 정상 동작하지 않는 것으로 추정됩니다.

원인 분석: 로그인 실패의 근본 원인은 **인증 데이터 또는 절차의 문제**로 볼 수 있습니다. 다음 가능한 요인을 점검했습니다:

- **초기 사용자 데이터 없음:** 시스템에 기본 계정(admin 등)이 존재하지 않는 경우 로그인 시 항상 "아이디 또는 비밀번호가 잘못되었습니다" 오류가 발생합니다. 백엔드 `login_for_access_token` 구현을 보면, 주어진 login_id로 사용자 조회 후 비밀번호 검증을 합니다 [25]. 만약 해당 login_id가 DB에 없거나 비밀번호 해시가 맞지 않으면 401 오류를 반환합니다. 이 프로젝트에서는 최초 실행 시 `/api/init` 엔드포인트를 통해 기본 계정을 생성하도록 되어 있습니다 [4] [26]. 즉, **초기화를 하지 않고** 곧바로 로그인하면 당연히 실패합니다. Docker 환경에서는 compose 설정상 backend 서비스가 기동될 때 자동으로 `/api/init`이 호출되지 않으므로, 사용자가 수동으로 초기화를 해주어야 합니다. 이 절차가 누락되었을 가능성이 큽니다 (예: 운영 환경 배포 시 init를 깜빡한 경우).

해결책: **시스템 초기화 수행** – 관리자/간사/평가위원 기본 계정을 DB에 만들어주는 `/api/init` 엔드포인트를 실행해야 합니다. 해당 엔드포인트는 최초 한 번만 성공하며, 이미 유저가 있을 경우 "이미 초기화되었습니다" 메시지를 반환합니다 [27]. 초기화 시 `admin/admin123`, `secretary01/secretary123`, `evaluator01/evaluator123` 등의 계정이 생성되며 [28] [29], 이후 이 계정들로 로그인을 시도해야 정상적으로 토큰이 발급됩니다.

- **비밀번호 해싱/검증 문제:** 기본 계정 생성 시 비밀번호는 평문으로 입력되지만 DB에 저장될 때 이미 해시되어 저장됩니다 [30]. 로그인 절차에서는 `verify_password(plain, hashed)` 함수를 통해 검증하며, 이는 `passlib`의 bcrypt 컨텍스트를 사용하므로 정상 동작해야 합니다 [31]. 만약 **해싱 알고리즘 불일치**나 잘못된 해시가 저장되면 검증 실패합니다. 그러나 코드상 `/api/init`에서 `get_password_hash`로 해시하여 저장하므로, 알고리즘 불일치는 없을 것으로 보입니다. 따라서 해시 이슈 가능성은 낮습니다.

- **인증 요청 형식 불일치:** 프론트엔드에서 로그인 API 호출 시 **요청 데이터 포맷**이 잘못되면 인증이 실패할 수 있습니다. FastAPI의 `/auth/login` 경로는 `OAuth2PasswordRequestForm`을 사용하므로 `username`과 `password` 필드를 **폼 데이터(form-data)**로 받아야 합니다 [32]. 프론트엔드 Axios 요청을 확인한 결과, `FormData` 객체에 `username`과 `password`를 담아 `multipart/form-data`로 POST하고 있어 형식은 올바릅니다 [33]. (프론트엔드 `App.js`에서 `axios.post(API+'/auth/login', formData, {headers: {'Content-Type': 'multipart/form-data'}})` 형식으로 요청함 [33].) 따라서 **요청 자체는 정상**으로 보입니다.

- **CORS 또는 클라이언트-서버 환경 문제:** 로그인이 아예 시도되지 못하고 에러가 난다면 CORS 차단이나 서버 URL 오설정이 원인일 수 있습니다. 그러나 `.env` 및 Docker 설정을 보면, 프론트엔드 환경변수 `REACT_APP_BACKEND_URL`이 `http://localhost:8080`으로 맞춰져 있고 [34], 백엔드도 CORS 허용 도메인에 `http://localhost:3000` (또는 3001) 등을 포함하고 있습니다 [35]. 과거 기본 설정에서 `3001`만 허용되었으나 .env에서 `3000`도 추가된 것으로 확인됩니다 [36]. Docker-Compose에서는 명시적으로 `CORS_ORIGINS`에 `http://localhost:3000`을 포함시켜 백엔드 서비스에 주입하고 있습니다 [35]. 따라서 현재 CORS 설정은 적절하며, 프론트엔드(포트 3000/3001)에서 백엔드(8080) 호출이 차단되지는 않을 것입니다. (만약 누락 시 해당 환경변수를 추가 설정해야 합니다.)

- **기타 API 불안정 요소:** 서버 측 `SecurityMiddleware`나 `IPWhitelistMiddleware`가 인증 흐름을 방해할 가능성도 있습니다. `IPWhitelistMiddleware`는 `/api/admin` 및 `/api/init` 경로에 IP 제한을 둘 수 있는데 [37], 만약 기본값이 로컬호스트만 허용이라면 다른 IP의 요청을 차단했을 수 있습니다. Docker 환경에서 프론트엔드 컨테이너의 IP가 백엔드와 다르므로, **간사 승인 관련 API**나 **초기화 API**가 차단되었을 여지가 있습니다. 이 경우 관리자 페이지에서 간사 승인(API가 `/api/admin/...`)이 실패하거나, init API 호출이 실패했을 수 있습니다. 해결책은 개발 중에는 IP 화이트리스트를 비활성화하거나(`IPWhitelistMiddleware` 미적용), 허용 IP에 프론트엔드 컨테이너 네트워크 대역을 추가하는 것입니다. 운영 환경에서는 보안을 위해 제한을 두되, VPN 또는 특정 서버 IP만 허용하도록 설정해야 합니다.

개선/보완 계획:

1. **초기 계정 자동 등록** – 사용자 개입 없이도 기본 계정이 보장되도록 개선합니다. 예를 들어, 서버 start 이벤트(`@app.on_event("startup")`)에서 `db.users` 컬렉션에 관리자 계정이 존재하는지 확인하고, 없으면 내부적으로 admin 계정을 생성하도록 할 수 있습니다. 하지만 이러한 자동 생성은 운영 시 의도치 않게 동작할 수 있으므로, 대신 **초기화 절차를 문서화**하고 운영 단계에서 반드시 `/api/init`을 호출하도록 가이드하는 것이 현실적입니다. Docker 환경에서는 `depends_on` 이후 일정 시간 지연 후 init을 자동 호출하는 스크립트를 추가하거나, docker-compose에 주석 처리된 `user_creator` 서비스를 활용해 초기화를 할 수도 있습니다 (예: `user_creator` 컨테이너가 기동되어 `create_test_user.py` 등을 실행하도록) [38]. 현재 `/api/init` 엔드포인트가 이미 구현되어 있으므로 이를 적극 활용하도록 합니다.

2. **환경 설정 재검증** – `.env` 및 compose 설정에서 인증 관련 변수를 재점검합니다. JWT 시크릿 키(`JWT_SECRET_KEY`)가 dev용으로 설정되어 있는데 운영 시 교체 필요하고, Access Token 만료 시간 등도 30분으로 설정되어 있습니다 [39]. 만약 토큰 만료로 인해 재로그인이 필요한 상황이라면 리프레시 토큰 발행 등 추가 기능이 필요하나, 현재는 로그인 자체가 안되는 문제가 우선이므로 해당 부분은 추후 고려합니다. 또한 CORS 허용은 현재 로컬 개발 주소들만 등록되어 있으므로, 실제 서비스 도메인(예: `https://evaluation.myservice.com`) 등을 허용 목록에 추가해야 합니다 (운영 배포 시 환경변수로 지정).

3. **로그/모니터링 활성화** – 로그인 실패 원인을 추적하기 위해 서버 로그를 확인하거나, `login_for_access_token` 함수에 추가 로깅을 넣는 것도 도움이 됩니다. 현재 코드에서 실패 시 단순 401 예외만 던지므로 [25], 예를 들어 잘못된 로그인 시도를 일정 횟수 모니터링하거나, 어떤 login_id로 시도되었는지 정도를 경고 로그로 남겨두면 원인 파악에 유용합니다.

4. **UI/UX 개선** – 프론트엔드에서 로그인 실패 시 사용자에게 표시되는 메시지를 명확히 하고 (현재 아마도 "아이디 또는 비밀번호가 잘못되었습니다"라는 백엔드 응답을 그대로 보여줄 가능성이 있음 [40]), 초기화를 안 한 상황이라면 "시스템 관리자 계정이 설정되지 않았습니다" 같은 안내를 할 수 있도록 하면 좋습니다. 물론 이는 부가적인 제안이며, 핵심은 **백엔드 인증 흐름의 안정성 확보**입니다.

이러한 조치를 통해 관리자/간사/평가위원 계정 로그인 문제가 해결될 것입니다. 특히 **초기 데이터 준비**와 **환경설정 보완**이 가장 중요한 부분입니다. 실제 `/api/init` 실행 후에는 `admin/admin123` 등으로 정상 로그인되어 JWT 토큰과 사용자 정보가 반환되는 것을 확인할 수 있습니다 [41] [42].

추가로, 간사 회원가입 승인 흐름 등도 점검해야 합니다. 예컨대 간사 승인시 새로운 User 생성 후 해당 계정으로 로그인 테스트, 평가위원 계정 생성 후 로그인 테스트 등을 수행해 **전반적인 인증 시나리오**를 테스트하여 안정화합니다.

4. 프론트엔드 연동 오류 – 백엔드 API 불일치로 기능 미작동

문제 현상: 프론트엔드에서 백엔드 API 연동이 원활하지 않아, 특히 **평가 템플릿 관리** 등의 기능이 동작하지 않습니다. 사용자가 웹 UI에서 수행하는 조작에 대해 백엔드 응답이 없거나 오류가 발생합니다. 예를 들어, 템플릿 목록 조회/생성, 평가표(템플릿) 수정 등의 시도가 실패하는 상황입니다.

원인 분석: 이 문제는 **백엔드에 구현된 API 경로/형식이 프론트엔드의 예상과 불일치**하거나, 아예 **백엔드에 해당 기능이 미구현**되어 발생합니다. 구체적으로, `frontend/src/components/TemplateManagement.js` 코드를 분석해 보면 프론트엔드가 기대하는 API와 동작을 확인할 수 있습니다:

- 템플릿 목록 조회: `GET /api/templates?project_id=...` 형태로 프로젝트별 템플릿 목록을 가져옵니다 [43]. 백엔드에도 해당 경로가 구현되어 있습니다: `@api_router.get("/templates")`가 존재하며, `project_id`를 쿼리 파라미터로 받아 필터링합니다 [44]. **문제는** 백엔드 함수 시그니처가

`project_id: Optional[str] = None`로 정의되어 있어 query로 받도록 한 반면, 프론트에서는 Axios `params`로 `project_id`를 전달하고 있어 호환됩니다. 이 부분은 비교적 정상 동작할 것입니다.

- 템플릿 생성: `POST /api/templates`로 새 템플릿 데이터를 보냅니다 [45]. 프론트에서는 `templateData` 객체에 `name, description, items` 등을 담고 `project_id`도 함께 JSON 바디로 보내고 있습니다 [45]. 그러나 백엔드 구현을 보면 `create_template` 함수가 `project_id: str`를 **함수 인자**로 별도로 받고, 바디에는 `template_data: EvaluationTemplateCreate`만 받도록 되어 있습니다 [24]. FastAPI에서는 함수 인자가 경로나 쿼리 매개변수로 취급되므로, 현재 코드에 따르면 `project_id`는 **쿼리 스트링**으로 전달되어야 합니다. 프론트엔드는 이를 인지하지 못하고 JSON에 포함시켰기 때문에, 백엔드에서는 `project_id` 인자를 받지 못해 **HTTP 422 오류**(요청 검증 실패)가 발생할 것입니다. 요컨대 **템플릿 생성 API의 파라미터 처리 불일치**가 있습니다.

- 템플릿 상세 조회: 프론트에서는 `GET /api/templates/{template_id}`로 호출하고 있습니다 [46]. 하지만 백엔드에는 해당 경로에 대한 엔드포인트가 구현되지 않았습니다 (검색해본 결과 `/templates/{id}` GET 함수가 없음). 따라서 404 에러가 발생할 것입니다. 이는 **기능 미구현** 사례입니다.

- 템플릿 수정/삭제: 프론트에서 `handleUpdateTemplate` 등이 `PUT /api/templates/{id}` 또는 `DELETE /api/templates/{id}`를 호출하도록 예상됩니다 [47] [48] (주석에 계획이 명시되어 있음). 그러나 이 역시 백엔드에 엔드포인트 부재로 동작하지 않습니다.

- 템플릿 공유 등 고급 기능: `POST /api/templates/{id}/share` 등도 주석상 존재하지만, 구현이 안 되어 있습니다 [49]. 프론트 UI에서 공유 모달을 열고 사용자 입력을 받더라도, 백엔드 연동이 없으므로 실패합니다.

요약하면, **백엔드와 프론트엔드 간 API 계약 불일치**와 **백엔드 기능 미완성**이 주요 원인입니다. 이는 앞서 살펴본 `TODAY_WORK_SUMMARY.md`에서도 백엔드의 해당 함수들이 85%만 구현되고 내일 마무리 예정이었다고 언급되어 있습니다 [13] [14]. 즉, 아직 끝나지 않은 부분이 프론트에는 구현된 상황입니다.

해결 방안: 프론트엔드와 백엔드의 **API 사양을 일치**시키고, 빠져있는 백엔드 기능을 구현해야 합니다. 구체적인 대응은 다음과 같습니다:

1. **API 인터페이스 정합성 수정:** 예를 들어 **템플릿 생성 API**는 프론트엔드가 보내는 방식에 맞춰 백엔드를 수정하는 것이 간편합니다. `create_template` 함수의 시그니처를 `template_data: EvaluationTemplateCreate`에 `project_id` 필드를 포함시키도록 바꾸거나, 아예 EvaluationTemplateCreate 모델에 `project_id: str`를 넣는 방법이 있습니다. 전자가 빠르다면 아래처럼 할 수 있습니다:

```
@api_router.post("/templates", response_model=EvaluationTemplate)
async def create_template(template: EvaluationTemplateCreate, current_user: User = Depends(...)):
    project_id = template.project_id  # 모델 내 필드로부터 추출
    ...
```

그리고 EvaluationTemplateCreate 모델을 `name, description, items, project_id` 필드로 정의합니다. **대안**으로, 프론트엔드에서 `axios.post` 호출시 `url: /templates?project_id=${projectId}` 형태로 쿼리스트링을 추가해 백엔드 기존 구현에 맞추는 방법도 있습니다. 하지만 일반적으로 REST API 설계 상 **POST 본문에 리소스 필드를 포함**하는 편이 자연스럽고, 프론트 수정보다 백엔드 수정을 권장합니다.

2. **누락된 템플릿 API 구현:** 템플릿 관리 기능이 제대로 작동하려면 아래 백엔드 엔드포인트들을 추가로 구현해야 합니다 (프론트 요구사항에 따라):

3. `GET /api/templates/{template_id}` – 특정 템플릿 상세 조회. `db.evaluation_templates.find_one({"id": template_id})`로 찾은 후 `EvaluationTemplate` 모델로 리턴.
4. `PUT /api/templates/{template_id}` – 템플릿 수정. 요청 바디로 수정 내용(이름, 설명, 항목 등)을 받고 DB에서 해당 템플릿 문서 업데이트.
5. `DELETE /api/templates/{template_id}` – 템플릿 삭제. 실제로는 status를 'archived'로 바꾸는 등 소프트 딜리트 구현이 바람직.
6. `POST /api/templates/{template_id}/share` – 템플릿 공유. 요청 바디로 `user_id`와 `permission`을 받아 공유 정보를 저장. (공유 기능 구현을 위해 별도의 컬렉션이나 `evaluation_templates` 문서에 공유자 목록 필드 추가 등을 해야 함.)
7. `POST /api/templates/{template_id}/clone` – 템플릿 복제. 지정 템플릿을 복사하여 새 템플릿 문서 생성.
8. `PATCH /api/templates/{template_id}/status` – 템플릿 상태 변경. (예: draft->active 등)

이러한 엔드포인트들은 TemplateManagement 컴포넌트의 주석에 상세히 나열되어 있습니다 [48] [50]. 프론트엔드에서 해당 기능 UI를 표시하고 있으므로, 백엔드도 구현을 완료해야 사용자가 정상 이용 가능합니다. 현재 백엔드에 있는 `/templates` 관련 엔드포인트는 목록 조회와 생성 두 개뿐이며, 그것도 완전히 다듬어지지 않은 상태입니다 [44] [24]. **우선순위**로 상세조회(`GET {id}`)와 수정(`PUT {id}`)을 구현하고, 여력이 되면 삭제/공유 등 추가 기능을 구현합니다.

1. **기타 연동 기능 점검:** 평가표(assignment) 관리, 평가 제출 등의 기능도 확인이 필요합니다. 예컨대 프론트 `EvaluationManagement.js`나 기타 컴포넌트에서 사용하는 `/api/assignments`나 `/api/evaluations` 관련 호출이 백엔드에 있는지 살펴봐야 합니다. 코드에 `/assignments` 관련 POST 구현은 일부 존재하나 (배치 할당 등) 완벽하지 않을 수 있습니다 [51] [52]. 이러한 부분도 프론트와 조율하여 완성시켜야 합니다. 프론트엔드 개발자와 협업하여 **API 스펙 문서**를 작성/업데이트하고, 누락된 API를 채워넣는 과정이 필요합니다.

2. **프론트엔드 임시 대응:** 백엔드가 즉시 수정되기 어려운 경우, 프론트엔드에서 임시로 대응할 수 있는 부분도 있습니다. 예를 들어 앞서 언급한 템플릿 생성의 `project_id` 문제는 프론트에서 쿼리스트링으로 붙여 보내는 식으로 바꿔 일단 동작시키고, 백엔드 수정 시 다시 돌리는 방법이 있습니다. 그러나 장기적으로 유지보수 관점에서 백엔드 수정이 바람직하므로, 임시 대응은 최소화합니다.

기대 효과: 위 개선이 이루어지면, 프론트엔드 **평가표/템플릿 관리 UI**에서 조회/생성/수정/삭제 등의 작업이 오류 없이 진행됩니다. 예를 들어, 사용자가 새 평가 템플릿을 생성하면 백엔드 `/api/templates`가 정상 처리하여 DB에 템플릿이 저장되고, 응답으로 생성된 템플릿 객체를 반환할 것입니다 [53] [9]. 이어서 프론트엔드는 목록을 재조회하여 방금 생성한 템플릿이 나타나는 것을 확인할 수 있습니다 [54] [55]. 또한 템플릿 상세보기나 편집 시에도 `/api/templates/{id}` GET/PUT이 구현되어 있으므로, 화면에 상세 내용이 표시되고 수정사항이 반영됩니다.

나아가 프론트엔드와 백엔드의 협업 개발에서는, 이러한 **스펙 불일치 문제를 사전에 조율**하는 것이 중요합니다. Swagger docs (`/docs`)나 별도 API 명세서(`기술_명세서.md` 등)를 통해, 양측이 같은 기준으로 개발하도록 하고, 만약 변경이 생기면 양쪽 코드를 동기화해야 합니다. 현재 발견된 연동 오류는 대부분 이 통신 사양 어긋남에서 기인하므로, 수정 후에는 **종합 테스트**를 통해 다른 부분은 없는지 점검해야 합니다.

5. 도커 실행 환경 점검 및 개선 – 컨테이너화 및 배포 안정성

현황 요약: 제공된 Dockerfile 및 docker-compose 설정을 검토한 결과, 전반적으로 **MongoDB, Redis, Backend, Frontend**를 포함한 컨테이너 구성이 잘 이루어져 있습니다 [56] [57]. 개발(로컬) 환경과 운영 환경 모두 지원하도록 multi-stage Dockerfile과 여러 compose 파일이 준비되어 있으며, 헬스체크와 볼륨 마운트도 설정되어 있습니다 [58] [59]. 그러나 앞서 지적된 문제들로 인해 Docker 실행 시 완벽히 동작하지 않을 여지가 있습니다. 이를 해결하고 컨테이너 기반 배포를 안정화하기 위한 개선 사항을 제안합니다:

- **상대 경로 임포트 문제 해결:** 가장 시급한 것은 **이슈 1의 ImportError**를 Docker 환경에서도 해결하는 것입니다. 현재 `docker-compose.yml`에서 백엔드 서비스는 `command: python -m uvicorn server:app ...`로 어플리케이션을 실행합니다 [60]. 앞서 분석했듯 이 방식은 `server.py`의 상대 임포트를 실패하게 만듭니다. Dockerfile에서 `ENV PYTHONPATH=/app`로 설정하였지만, 패키지로 인식되지 않는 한 한계가 있습니다 [7]. 따라서 **Docker 이미지 빌드 시에 임포트 경로 수정본이 반영**되어야 합니다. 코드 수정이 완료되면 새로운 이미지를 빌드하여 문제를 해결합니다. (이미 Dockerfile은 변경된 코드를 COPY하도록 되어 있으므로, 코드만 고치면 됩니다.)

- **DB 초기화 및 설정 일관성:** Compose 파일을 보면 MongoDB 컨테이너의 기본 DB 이름은 `online_evaluation`로 초기화하지만 [61], 백엔드에서는 `DB_NAME=evaluation_db`로 사용하고 있습니다 [62]. 다행히 Mongo는 사용 시 자동으로 DB를 생성하므로 큰 충돌은 없지만, 혼란을 줄이기 위해 **DB 이름을 통일**하는 것이 좋습니다. 예컨대 모두 `evaluation_db`로 사용하거나 모두 `online_evaluation`로 맞춥니다. 환경변수 및 Mongo URI에서 일관되게 수정하십시오. 이외에 Redis도 비밀번호가 없는 설정인데, 운영 시에는 Redis에 인증을 설정하고 환경변수(`REDIS_URL`에 비번 포함)를 추가하는 편이 안전합니다.

- **환경변수 및 시크릿 관리:** 현재 Docker-compose에 앱 환경변수가 노출되어 있습니다 [63]. 운영에서는 `.env` 파일을 docker-compose와 함께 사용하거나, Kubernetes 사용 시 시크릿으로 관리해야 합니다. 특히 JWT_SECRET_KEY 등은 코드/레포에 평문으로 두지 말고 안전하게 관리해야 합니다. 이 부분은 보안적인 개선 사항입니다.

- **컨테이너 간 네트워킹 및 CORS:** compose는 모든 서비스를 같은 네트워크에 넣고, 프론트엔드에서 `REACT_APP_BACKEND_URL=http://localhost:8080`로 백엔드에 접근하도록 하고 있습니다 [64]. docker-compose를 로컬에서 실행할 때는 frontend 컨테이너의 localhost:80이 호스트의 3001로, backend 컨테이너의 8080이 호스트의 8080으로 매핑됩니다. 프론트엔드 브라우저에서 API 요청시 `http://localhost:8080`으로 가면 이는 호스트(개발자 PC)의 8080을 의미하므로 백엔드 컨테이너에 잘 연결됩니다. 이 설정은 로컬 개발에 적합합니다. 운영에서는 Nginx를 사용해 80/443을 받도록 (production 프로파일) 설정되어 있습니다 [59]. Nginx 컨테이너가 frontend 정적 파일을 서빙하며, API 요청을 백엔드로 프록시하도록 `nginx.conf`가 설계되었을 것입니다. 따라서 **Nginx 설정**을 확인하여, 예컨대 `/api/` 경로는 백엔드(8080)으로 프록시되게 하고, 나머지는 정적 파일(프론트엔드 빌드 결과)로 제공하는지 봐야 합니다. 만약 누락되었다면 추가해야 합니다. 이로써 운영 시 단일 도메인에서 프론트+백엔드가 동작하여 CORS 문제가 자연히 해소됩니다. (로컬 개발 단계에서는 이미 CORS 허용 설정으로 해결한 상태입니다.)

- **의존 패키지 추가 (예: websockets):** 오늘 작업 요약에서 WebSocket 테스트 실패 원인으로 `websockets` 패키지 누락이 언급되었습니다 [65]. 실제 `requirements.txt`에 websockets가 없다면 FastAPI의 WebSocket 기능에는 영향 없지만, 만약 테스트 코드나 다른 모듈에서 이 패키지를 사용하려 한다면 import 에러가 날 것입니다. 그러므로 `backend/requirements.txt`에 `websockets`를 추가하고 이미지를 다시 빌드해야 합니다. `Dockerfile.backend`에서는 requirements.txt 설치 후 pytest나 flake8 등을 설치하고 있으므로 [66], 의존성 누락이 없도록 최신 requirements를 반영합니다.

- **로그 및 볼륨 권한:** compose에서 `/app/logs`를 호스트 볼륨으로 마운트하고 있습니다 [67]. 간혹 호스트-컨테이너 권한 문제로 로그파일이 안 써지는 이슈가 있을 수 있습니다. Dockerfile.production 단계에서 `appuser`로 권한 변경을 하며 logs 디렉토리에 777 권한을 주고 있어 크게 문제는 없어 보입니다 [68] [69]. 다만 개발용 (reload 사용) 컨테이너에서는 root로 동작하므로 볼륨 퍼미션 문제는 없을 것입니다. 종합적으로 현재 설정은 적절합니다.

- **CI/CD 및 배포 테스트:** Docker 구성요소 수정 후 실제 컨테이너 빌드와 실행 테스트를 수행해야 합니다. 데이터베이스 마이그레이션이나 시드(seed) 데이터 입력 (`/api/init`) 절차도 포함해서 자동화할 수 있다면 해봅니다. `docker-compose up`으로 모든 서비스가 올라간 뒤, `curl -f http://localhost:8080/health` 등의 헬스체크를 확인하고 [70], `curl -X POST http://localhost:8080/api/init`으로 기본계정 생성, `curl -X POST http://localhost:8080/api/auth/login -F username=admin -F password=admin123`으로 토큰 발급 등 일련의 과정이 문제없이 돌아가는지 확인합니다. 이러한 과정을 스크립트화하거나 README에 Usage로 문서화해 두면, 이후 운영 배포 시 시행착오를 줄일 수 있습니다.

정리하면, Docker 기반 실행을 위해 **코드 수정사항을 반영한 재빌드**, **초기화/시드 절차의 자동화 또는 가이드 마련**, **환경변수 정리**, **보안 설정 강화** 등이 이루어져야 합니다. 다행히 현재 Dockerfile과 Compose는 비교적 완성도가 높고, 지난 작업 내역에 따르면 모든 서비스가 Docker에서 정상 동작 확인까지 마친 상태입니다 [71]. 남은 것은 위의 버그 수정을 포함한 마지막 코드 수정을 반영하여 이미지를 업데이트하는 일입니다. 이를 통해 개발 환경뿐 아니라 운영 환경에서도 **컨테이너화된 온라인 평가 시스템이 일관되게 구동**되도록 할 수 있습니다.

결론 및 요약

이번 분석을 통해 **온라인 평가 시스템**의 백엔드 코드 구조와 프론트엔드 연동 현황을 살펴보고, 다섯 가지 핵심 문제에 대한 원인과 해결책을 도출하였습니다. 아래는 각 이슈별 조치 사항 요약입니다:

- **ImportError (상대 임포트)** – 모듈 임포트 경로를 절대경로로 수정하거나 패키지 구조를 갖추도록 개선하여 패키지 인식 오류를 해결함 [5]. 예: `from backend.cache_service import cache_service`로 수정.

- **모델/함수 정의 추가** – 누락된 Pydantic 모델(`FileMetadata` 등)과 유틸 함수(`generate_evaluator_credentials`, `update_project_statistics` 등)를 구현하여 참조 오류 제거. 모델 필드와 함수 로직은 코드 용도에 맞게 작성하고, 필요한 곳에 임포트함 [8] [72]. 이로써 평가위원 생성, 프로젝트 통계 업데이트, 파일 업로드 등의 기능이 정상 동작하게 됨.

- **로그인 실패 문제 개선** – `/api/init` 엔드포인트를 통한 **기본 계정 생성** 절차를 확실히 하고, 초기화 누락 시 로그인 불가함을 명시적으로 인지/대응. 환경변수 및 CORS 설정을 재확인하여 프론트엔드 요청이 차단되지 않도록 함 [35]. IP 화이트리스트 등 보안 미들웨어 설정도 개발 단계에서는 완화하거나 올바르게 설정하여, 불필요한 인증 실패를 방지.

- **프론트엔드-백엔드 API 정합화** – 프론트엔드가 호출하는 모든 API 경로에 대해 백엔드 구현을 갖추고, 요청/응답 데이터 형식을 일치시킴. 특히 템플릿 관리 관련 API들을 추가 구현하고 [43] [24], 쿼리파라미터 vs 바디 불일치는 백엔드 쪽으로 수정하여 프론트 수정 최소화. 구현이 완료되면 통합 테스트를 수행하여 UI 기능(템플릿 생성/조회/수정/삭제 등)이 기대대로 작동하는지 확인.

- **Docker 컨테이너 실행 안정화** – 수정된 코드를 기반으로 Docker 이미지를 재빌드하고, compose 설정을 미세 조정하여 (**상대임포트 문제 해결, 환경변수 정리, 초기화 절차**) 컨테이너 구동 시 오류가 없도록 함. 필요시

운영용 Nginx 설정 및 CI 파이프라인도 점검. 이로써 개발 머신 뿐 아니라 어떤 환경에서든 `docker-compose up -d`만으로 백엔드+프론트엔드+DB+캐시가 모두 기동되고 정상 동작하게 하는 것이 목표입니다.

마지막으로, 전체 코드의 구조와 흐름을 그림으로 표현하면 다음과 같습니다:

- **클라이언트 (브라우저)** ← (HTTP/WS) → **프론트엔드 컨테이너 (React 웹 서버)** ← (프록시/Nginx) → **백엔드 컨테이너 (FastAPI)** ↔ **MongoDB 컨테이너** (데이터 저장) 및 **Redis 컨테이너** (캐시 세션)
- 백엔드 `server.py`는 라우터별로 인증/권한을 체크하고 DB와 상호작용하여 JSON 결과를 반환. 프론트엔드는 Axios로 `/api/...` 요청을 보내고, 응답 데이터를 화면에 반영.
- JWT 인증 토큰은 로그인 시 발급되어 로컬스토리지 등에 저장되며, 이후 요청의 `Authorization: Bearer` 헤더로 첨부되어 백엔드에서 신원 검증을 합니다 [73].
- Admin/Secretary/Evaluator 각 역할에 따라 접속 가능한 화면과 API가 달라지며, 백엔드에서 `get_current_user`와 역할검증(`check_admin_or_secretary` 등) 미들웨어로 보호하고 있습니다 [74] [75].

이번 조치 계획을 이행함으로써, 시스템은 **보다 완전한 기능 구현과 안정적인 운영**이 가능해질 것입니다. 남은 세부 기능 구현과 테스트를 완료하면, 요구된 100% 완성도에 도달할 것으로 기대됩니다. 각 수정사항은 코드 레벨에서 꼼꼼히 적용하고, 커밋/배포 전에 통합 테스트를 돌려 회귀(regression)가 없는지 확인해야 합니다. 이상으로 GitHub 저장소 코드 분석 및 문제 해결 방안에 대한 보고를 마칩니다. [13] [14]

---

[1] [13] [14] [65] [71] TODAY_WORK_SUMMARY.md
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/TODAY_WORK_SUMMARY.md

[2] [3] [34] [35] [38] [56] [57] [58] [59] [60] [61] [62] [63] [64] [67] [70] docker-compose.yml
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/docker-compose.yml

[4] [5] [6] [8] [9] [17] [18] [19] [20] [21] [22] [24] [25] [26] [27] [28] [29] [30] [32] [37] [40] [41] [42] [44] [51] [52] [53] [72] [74] [75] server.py
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/backend/server.py

[7] [66] [68] [69] Dockerfile.backend
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/Dockerfile.backend

[10] [11] [12] [15] [16] [23] models.py
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/backend/models.py

[31] security.py
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/backend/security.py

[33] [73] App.js
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/frontend/src/App.js

[36] [39] .env
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/backend/.env

[43] [45] [46] [47] [48] [49] [50] [54] [55] TemplateManagement.js
https://github.com/bruce0817kr/Online-evaluation/blob/2f1462f8c92da0bd6a067620ed19ae0e08bdae7d/frontend/src/components/TemplateManagement.js