ChatGPT

온라인 평가 시스템 주요 기능 설계 및 구현 방안

온라인 평가 플랫폼 **Online-evaluation** 프로젝트의 미구현 핵심 기능들을, 해당 프로젝트의 기술 스택(프론트엔드는 React, 백엔드는 FastAPI 기반 [1])에 맞추어 모듈별로 설계하고 구현 방안을 제안합니다. 이 시스템은 **다중 역할(Role) 기반 접근 제어**를 적용하며, 관리자(Admin), 간사(Manager), 평가위원(Evaluator) 세 가지 사용자 권한에 따라 기능 접근이 제한됩니다 [2]. 아래에서는 요구된 각 기능에 대해 프론트엔드 컴포넌트와 백엔드 API를 중심으로 설계 방안을 상세히 설명합니다.

1. 평가 개설 기능 (Admin/간사용)

기능 개요: 관리자 또는 간사가 새로운 평가를 개설하여 평가 **프로젝트**를 생성할 수 있습니다. 평가 제목, 설명, 대상 기업 리스트 등의 정보를 입력하고 평가를 생성하면, 데이터베이스에 새로운 평가 엔터티(프로젝트)가 저장됩니다. 이 **프로젝트**는 이후 평가위원 배정과 평가 점수 입력의 기반이 됩니다 [2].

- **프론트엔드 컴포넌트:** 관리자가 접속하는 프론트엔드에는 `CreateEvaluationPage` 또는 `EvaluationForm` 컴포넌트를 구현합니다. 이 폼에는 **평가 제목**, **평가 설명**, **대상 기업 목록** 등을 입력하는 필드가 포함됩니다. 대상 기업 목록은 콤마로 구분된 기업명 입력이나 엑셀 업로드로 처리할 수 있습니다. 예를 들어, React 훅을 활용하여 폼 상태를 관리합니다:

```
function CreateEvaluationForm() {
  const [title, setTitle] = useState("");
  const [description, setDescription] = useState("");
  const [companiesText, setCompaniesText] = useState("");  // 줄바꿈이나 쉼표로 구분된 회사 목록

  const handleSubmit = async (e) => {
    e.preventDefault();
    const companies = companiesText.split(/\r?\n/);  // 개행 기준으로 기업명 배열 생성
    const payload = { title, description, companies, criteria: /* 아래에서 설정 */ };
    // JWT 포함 헤더와 함께 POST 요청 전송 (관리자/간사 권한 필요)
    await fetch("/api/evaluations", {
      method: "POST",
      headers: { Authorization: `Bearer ${authToken}`, "Content-Type": "application/json" },
      body: JSON.stringify(payload)
    });
    alert("평가가 생성되었습니다.");
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* 제목, 설명 입력필드와 기업 목록 텍스트에리어 등 */}
    </form>
  );
}
```

위 컴포넌트는 관리자나 간사 계정으로 로그인했을 때만 접근 가능하도록 **Route 가드**를 적용합니다. 예를 들어 React Router에서 `<AdminRoute>` 컴포넌트를 만들어 현재 사용자 권한이 admin 또는 manager(간사)인지 확인하고, 아니라면 접근을 막습니다.

- **백엔드 API:** 새로운 평가 프로젝트 생성을 위한 엔드포인트를 설계합니다. 예를 들어 `POST /api/evaluations` 경로에 **관리자 또는 간사 전용** API를 구현합니다. FastAPI의 **의존성 주입(Dependency)** 기능을 이용하여 JWT 인증 및 역할 검증을 수행합니다.

```
from fastapi import Depends, HTTPException, status
from .auth import get_current_user  # JWT에서 현재 사용자 추출
from .models import EvaluationCreate, User  # Pydantic 모델 (스키마 정의)

def admin_or_secretary(current_user: User = Depends(get_current_user)):
    # 관리자가 아니고 간사도 아니면 예외 발생
    if current_user.role not in ("admin", "secretary"):
        raise HTTPException(status_code=403, detail="권한이 없습니다.")
    return current_user

@app.post("/api/evaluations", dependencies=[Depends(admin_or_secretary)])
async def create_evaluation(eval_data: EvaluationCreate):
    # 평가 데이터 구성
    new_eval = {
        "title": eval_data.title,
        "description": eval_data.description or "",
        "companies": eval_data.companies,          # 기업 리스트
        "criteria": eval_data.criteria,            # 평가 항목 (후술)
        "created_by": eval_data.created_by,        # 생성자 ID
        "created_at": datetime.utcnow(),
        "status": "draft"
    }
    result = await db.evaluations.insert_one(new_eval)  # MongoDB 예시
    return {"evaluation_id": str(result.inserted_id), "status": "created"}
```

위 코드에서 `admin_or_secretary` 디펜던시는 JWT로부터 얻은 현재 사용자 정보를 검사하여 관리자 또는 간사 권한이 아닌 경우 요청을 거부합니다. `create_evaluation` 엔드포인트는 전달된 평가 제목, 설명, 기업 목록 등을 바탕으로 데이터베이스에 평가 기록을 추가합니다. 예시에서는 MongoDB 사용을 가정하여 `insert_one`을 호출하였지만, 프로젝트에서 사용 중인 ORM(SQLAlchemy)이나 DB 스키마에 맞춰 동일하게 신규 **평가 엔티티**를 생성하면 됩니다 [3]. 데이터 모델 측면에서는 **Evaluation** 테이블/컬렉션에 `title`, `description`, `companies`(기업명 목록 혹은 회사 엔티티의 참조), `created_by`, `status` 등의 필드를 갖도록 설계합니다.

> **참고:** README에 따르면 본 시스템은 **프로젝트 기반 조직화** 개념을 갖고 있으며, 평가 템플릿을 프로젝트에 할당하는 구조로 확장될 수 있습니다 [2]. 단순화를 위해 여기서는 평가 생성 시 곧바로 평가 항목과 기업들을 포함하는 하나의 프로젝트를 만드는 것으로 설명합니다.

2. 평가 항목 및 점수 설정 기능

기능 개요: 평가를 개설할 때 **세부 평가 기준(항목)**과 각 항목의 배점, 가점 여부 등을 설정할 수 있어야 합니다. 이를 통해 평가 담당자는 유연한 평가 템플릿을 만들 수 있습니다 [4]. 예를 들어 “혁신성(30점)”, “사업성(20점)”, “추가 가점(5점, 가점)” 등의 항목을 정의할 수 있습니다.

- **프론트엔드 컴포넌트:** 평가 항목 설정은 평가 생성 폼 내에서 **동적 입력 필드 목록**으로 구현합니다. `CreateEvaluationForm`에서 "항목 추가" 버튼을 눌러 새로운 평가 기준 필드를 추가할 수 있고, 항목명, 만점 점수, 가점 여부 등을 입력하게 합니다. React 상태로 항목들의 배열을 관리하며, 입력값을 수집해 서버로 전송합니다. 예시 코드:

```
// CreateEvaluationForm 컴포넌트 내 일부
const [criteriaList, setCriteriaList] = useState([
 { name: "", maxScore: 0, bonus: false }
]);

const addCriterion = () => {
 setCriteriaList([...criteriaList, { name: "", maxScore: 0, bonus: false }]);
};

// criteriaList를 기반으로 폼 생성
return (
 <div>
  {criteriaList.map((crit, idx) => (
   <div key={idx}>
    <input
     type="text" placeholder="항목명"
     value={crit.name}
     onChange={e => updateCriterion(idx, { name: e.target.value })}
    />
    <input
     type="number" placeholder="배점"
     value={crit.maxScore}
     onChange={e => updateCriterion(idx, { maxScore: Number(e.target.value) })}
    />
    <label>
     <input
      type="checkbox"
      checked={crit.bonus}
      onChange={e => updateCriterion(idx, { bonus: e.target.checked })}
     />
     가점 여부
    </label>
   </div>
  ))}
  <button type="button" onClick={addCriterion}>+ 항목 추가</button>
 </div>
);
```

위와 같이 사용자는 여러 개의 평가 항목을 입력할 수 있으며, 각 항목별로 **배점**(만점 점수)과 **가점 여부**를 설정합니다. 가점 항목인 경우 총점 계산 시 별도로 취급하거나 최대 총점을 초과하여 부여될 수 있도록 백엔드에서 로직을 처리하게 됩니다. 프론트엔드는 입력된 기준들을 `criteriaList` 배열로 관리하다가 최종 **평가 생성** 제출 시 이 배열을 함께 API에 전송합니다.

- **백엔드 API 및 데이터 모델:** 앞서 `POST /api/evaluations` 엔드포인트에서 `criteria` 필드를 함께 받아 처리하도록 하였습니다. `EvaluationCreate` Pydantic 모델에 `criteria: List[Criterion]` 필드를 포함시켜, 각 Criterion에 `name`, `max_score`, `bonus` 등을 정의합니다. 서버에서는 이 리스트를 받아 그대로 DB에 저장하거나, 별도의 **평가항목 테이블**에 개별 저장할 수 있습니다. 간단한 구현으로는 평가 컬렉션/테이블 내에 JSON 배열로 항목들을 저장하고 평가 시 활용하는 방법이 있습니다 [3]. 예를 들면 MongoDB를 사용할 경우 아래와 같이 저장 가능합니다:

```
# Criterion 스키마 예시
from pydantic import BaseModel
class Criterion(BaseModel):
    name: str
    max_score: int
    bonus: bool = False

class EvaluationCreate(BaseModel):
    title: str
    description: Optional[str] = ""
    companies: List[str]
    criteria: List[Criterion]
    # ... 기타 필드
```

```
# create_evaluation 함수 내
new_eval = {
    "title": eval_data.title,
    "description": eval_data.description,
    "companies": eval_data.companies,
    "criteria": [c.dict() for c in eval_data.criteria],
    "created_by": current_user.id,
    "created_at": datetime.utcnow(),
    "status": "draft"
}
await db.evaluations.insert_one(new_eval)
```

위와 같이 **평가 항목(criteria)** 배열에 각 항목의 세부 정보가 포함됩니다. 추후 평가위원들이 점수를 제출할 때 이 `criteria` 정보를 기반으로 입력값의 검증 및 점수 합산을 수행합니다. 예를 들어, 각 항목별로 `max_score`를 넘지 않는지 검증하고, `bonus: true` 항목은 별도로 처리하여 총점 계산 시 가산점을 부여합니다.

> **유연한 점수 체계:** 본 시스템은 1-5점 척도, 백분율 등 다양한 점수체계를 지원할 수 있도록 설계되었습니다 [4]. 따라서 각 Criterion에 `type` 또는 `scale` 속성을 추가하여 "정수 1-5", "백분율(0-100)", "예/아니오" 등의 평가 방식을 지정할 수도 있습니다. 기본 예시에서는 정수 점수를 기준으로 설명합니다.

3. 평가위원 배정 기능

기능 개요: 생성된 평가 프로젝트에 대해 **평가위원을 배정**하는 기능입니다. 관리자는 해당 평가에 어느 평가위원들이 참여하여 평가할 것인지 지정할 수 있어야 합니다. 간사도 이 기능을 수행할 수 있으며(관리자와 유사 권한), 배정된 평가위원들만 해당 평가에 대한 점수 입력 권한을 갖습니다.

- **프론트엔드 컴포넌트:** 관리자/간사가 보는 **평가 상세 페이지** 또는 별도의 **배정 관리 페이지**에서 평가위원을 배정합니다. UI 상에서는 시스템에 등록된 **평가위원 계정 목록**을 확인하여 체크박스로 선택하거나, 드롭다운으로 여러 명을 선택할 수 있습니다. 예를 들어 `EvaluatorAssignModal` 컴포넌트를 띄워 현재 프로젝트에 배정된 평가위원들을 표시하고, 추가/제거를 할 수 있게 합니다. 사용자가 배정을 확정하면 API 호출이 일어나고 모달을 닫습니다.

- 사용자 흐름 예시:

  1. 관리자가 **평가 목록**에서 특정 평가를 선택하여 상세 화면으로 진입합니다.
  2. 상세 화면에서 "평가위원 배정" 버튼을 클릭하면, 현재 시스템에 **승인된 평가위원 목록**이 표시됩니다 (예: 체크박스 리스트).
  3. 관리자가 해당 평가에 배정할 평가위원들을 선택한 뒤 "배정 저장"을 누릅니다.
  4. 선택된 평가위원 IDs 목록이 백엔드 API로 전송되고, 성공 시 프론트엔드에서는 해당 평가에 배정된 평가위원 명단을 갱신합니다.

- **백엔드 API:** 평가위원 배정을 위한 엔드포인트를 설계합니다. 예를 들어 `POST /api/evaluations/{eval_id}/assign` 형태로 구현하며, 요청 바디에 `evaluator_ids: List[str]`를 포함시킵니다. 이 엔드포인트는 관리자와 간사만 호출 가능하도록 보호되어야 합니다 (`Depends(admin_or_secretary)` 재사용). 구현 방식은 두 가지 옵션이 있습니다:

- **평가 엔터티 내에 배정 필드 저장:** Evaluation 레코드(프로젝트)에 `assigned_evaluators` 필드를 두어 사용자 ID 배열을 저장합니다.
- **별도 배정 엔터티 사용:** `Assignment` 테이블/컬렉션을 만들어 (evaluation_id, evaluator_id) 쌍을 여러 건 저장합니다.

간단한 접근으로 첫 번째 방법을 채택해 설명하면, 백엔드에서 해당 평가 도큐먼트를 찾아 배정 필드를 업데이트합니다:

```
@app.post("/api/evaluations/{eval_id}/assign", dependencies=[Depends(admin_or_secretary)])
async def assign_evaluators(eval_id: str, payload: dict):
    evaluator_ids = payload.get("evaluator_ids", [])
    # 평가 존재 여부 검사
    evaluation = await db.evaluations.find_one({"_id": eval_id})
    if not evaluation:
        raise HTTPException(status_code=404, detail="평가를 찾을 수 없음")
    # 평가 도큐먼트에 평가위원 목록 업데이트
    await db.evaluations.update_one(
        {"_id": eval_id},
        {"$set": {"assigned_evaluators": evaluator_ids}}
    )
    return {"assigned_count": len(evaluator_ids)}
```

위 코드에서 `assigned_evaluators` 배열에 선정된 평가위원들의 사용자 ID를 저장하고 있습니다. Role 기반 접근 제어를 통해 관리자/간사만 이 작업을 수행할 수 있고, 백엔드에서는 전달된 ID들이 실제 **평가위원 역할의 사용자들**

인지 추가로 검증할 수 있습니다. (예: `db.users.find({"_id": {"$in": evaluator_ids}, "role": "evaluator"})`로 확인).

이후 평가위원이 자신의 대시보드에서 **할당된 평가 목록**을 불러올 때, 해당 사용자 ID가 `assigned_evaluators`에 포함된 평가들만 조회되도록 API (`GET /api/evaluations?assigned_to_me=1` 등)를 구성합니다. 이를 통해 평가위원은 자신이 배정받은 평가만 접근할 수 있습니다.

4. 평가 진행 및 점수 입력 기능 (평가위원용)

기능 개요: 평가위원은 배정된 평가에 대해 기업별로 평가 점수를 입력하고 제출할 수 있습니다. 각 평가위원은 자신에게 할당된 평가 목록과 각 평가의 대상 기업들을 확인하고, **항목별 점수**와 코멘트를 입력하여 평가를 완료합니다. 시스템은 입력된 점수를 저장하고 필요에 따라 자동 합산이나 통계 처리를 수행합니다 [4].

- **프론트엔드 컴포넌트:** 평가위원용 **대시보드** 페이지 (`EvaluatorDashboard`)를 구현하여, 해당 평가위원에게 할당된 모든 평가 프로젝트를 목록으로 보여줍니다. 각 항목에는 평가 제목과 진행 상태 (예: "5개 기업 중 2개 평가 완료") 등의 요약을 표시합니다. 평가위원이 특정 평가를 클릭하면, **기업별 평가 페이지**로 이동하거나 기업 리스트를 볼 수 있습니다.

기업별 평가 화면에서는 선택한 기업의 이름과 평가 **항목 리스트**가 폼 형태로 제시됩니다. 예를 들어 `EvaluationScoreForm` 컴포넌트를 사용하여 해당 기업에 대한 평가 항목별 입력 필드를 생성합니다. 각 항목의 이름과 배점(max_score)을 표시하고, 평가위원은 그 범위 내의 점수를 입력하거나 슬라이더를 움직여 점수를 정합니다. 또한 필요하면 각 항목 또는 전체 기업 평가에 대한 **코멘트** 입력란을 제공할 수 있습니다. 아래는 간략한 예시입니다:

```
function EvaluationScoreForm({ evaluation, company }) {
 const [scores, setScores] = useState({});
 const [comment, setComment] = useState("");

 const handleSubmit = async () => {
  const payload = { companyId: company.id, scores, comment };
  await fetch(`/api/evaluations/${evaluation.id}/submit`, {
   method: "POST",
   headers: { Authorization: `Bearer ${authToken}`, "Content-Type": "application/json" },
   body: JSON.stringify(payload)
  });
  alert(`${company.name} 평가 제출 완료`);
 };

 return (
  <form onSubmit={(e) => { e.preventDefault(); handleSubmit(); }}>
   <h2>{company.name} 평가</h2>
   {evaluation.criteria.map(item => (
    <div key={item.name}>
     <label>{item.name} (만점 {item.max_score}점)</label>
     <input
      type="number" max={item.max_score} min="0"
      value={scores[item.name] || 0}
      onChange={e => setScores({ ...scores, [item.name]: Number(e.target.value) })}
     />
    </div>
```

```
    ))}
    <div>
     <label>종합 코멘트:</label>
     <textarea value={comment} onChange={e => setComment(e.target.value)} />
    </div>
    <button type="submit">제출</button>
   </form>
  );
}
```

위 컴포넌트는 현재 평가의 항목 목록 (`evaluation.criteria`)에 따라 동적으로 입력 필드를 생성하고, `scores` 상태 객체에 항목명별 점수를 저장합니다. 제출 시 **POST 요청**을 통해 백엔드 API로 점수 데이터를 전송합니다. 프론트엔드는 제출 후 해당 기업을 사용자의 할당 목록에서 "완료" 표시하거나, 다음 기업 평가로 이동할 수 있게 UX를 설계합니다. 만약 평가 도중 임시 저장 기능(Progress Save)을 구현한다면, **onBlur** 이벤트나 "임시 저장" 버튼으로 백엔드에 중간 결과를 PUT/PATCH 요청하는 방안도 고려합니다.

- **백엔드 API:** 평가위원의 점수 제출을 처리하는 엔드포인트를 구현합니다. 위 예시에서는 `POST /api/evaluations/{eval_id}/submit` 형태로, 요청 바디에 `companyId`, `scores`, `comment` 등을 받습니다. 이 요청은 **평가위원 권한** 및 **배정된 사용자**만 호출 가능해야 하므로, `Depends(auth_required_evaluator)` 같은 의존성을 만들어 검사합니다. 검증 로직은 다음과 같습니다:
- JWT에서 추출된 `current_user`의 역할이 evaluator인지 확인.
- 해당 평가(eval_id)에 이 사용자가 배정되었는지 확인. (예: `evaluation.assigned_evaluators`에 user.id 포함 여부 검사)

검증 통과 시, 제출된 점수를 데이터베이스에 저장합니다. 데이터 모델은 **Submission** 컬렉션/테이블을 만들어 (`evaluation_id`, `company_id`, `evaluator_id`, `scores`, `comment`, `submitted_at`, `total_score` 등 필드) 한 건씩 추가하는 방법이 일반적입니다. 각 제출은 한 평가위원의 한 기업에 대한 평가 결과를 나타냅니다. (또는 `company_id`를 scores 딕셔너리에 포함시켜 한 번에 여러 기업을 제출하지 않는 한, 1개 기업당 1개 제출로 취급합니다.)

예시 구현 (MongoDB 사용 가정):

```
@app.post("/api/evaluations/{eval_id}/submit",
dependencies=[Depends(auth_required_evaluator)])
async def submit_scores(eval_id: str, submission: ScoreSubmitSchema, current_user: User =
Depends(get_current_user)):
    # 1. 역할 검증: auth_required_evaluator에서 evaluator임을 보장
    # 2. 배정 검증
    eval_doc = await db.evaluations.find_one({"_id": eval_id})
    if not eval_doc or current_user.id not in eval_doc.get("assigned_evaluators", []):
        raise HTTPException(status_code=403, detail="해당 평가에 대한 접근 권한 없음")
    # 3. 점수 저장 로직
    data = {
        "evaluation_id": eval_id,
        "company_id": submission.companyId,
        "user_id": current_user.id,
        "scores": submission.scores,            # {항목명: 점수} 또는 항목 ID: 점수
        "comment": submission.comment,
```

```
        "submitted_at": datetime.utcnow()
    }
    # 총점 계산 (가점 여부 반영하여 합산)
    total = 0
    for criterion in eval_doc["criteria"]:
        name = criterion["name"]
        max_score = criterion["max_score"]
        score = data["scores"].get(name, 0)
        if score > max_score:
            score = max_score  # 점수 상한 검증
        if criterion.get("bonus"):
            total += score  # 가점도 총점에 포함 (혹은 별도 처리 가능)
        else:
            total += score
    data["total_score"] = total
    await db.submissions.insert_one(data)
    return {"message": "submitted", "total_score": total}
```

위 로직은 제출된 점수를 DB에 저장하고 `total_score`를 계산하여 응답합니다. **가점(bonus)** 항목은 단순히 총점에 더해주거나, 필요시 별도 필드(`bonus_score`)로 분리해 관리할 수도 있습니다. `scores` 필드는 JSON으로 각 항목별 점수를 저장하며, 항목이 식별자(ID)로 관리된다면 키로 ID를 쓰는 게 바람직하지만, 간단하게 항목명을 키로 사용하고 있습니다.

또한 **중복 제출 방지/수정** 전략도 고려해야 합니다. 일반적으로는 한 평가위원이 같은 기업에 대해 한 번만 제출 가능하도록 하고, 제출 후에는 해당 평가를 **읽기 전용**으로 보여주거나 수정 기능은 제한합니다. 이러한 비즈니스 규칙은 요구사항에 맞게 추가 구현합니다.

백엔드에서는 평가 결과를 모아서 분석하거나 **실시간 진행률 대시보드**에 반영할 수 있습니다. (예: 몇 명의 평가위원이 제출을 완료했는지 등). 제출이 일어날 때마다 평가 완료 상태를 집계하거나, 클라이언트가 `GET /api/analytics/progress`를 호출해 실시간 진행 상황을 볼 수 있게 할 수 있습니다 [5] [6].

5. 회원 가입 및 승인 기능 (관리자 승인 제도)

기능 개요: 평가위원 또는 간사 역할의 사용자가 시스템에 회원가입을 하면, 바로 활성화되는 것이 아니라 **관리자 승인을 받아야만** 최종 로그인 및 시스템 사용이 가능하도록 합니다. 이는 민감한 평가 시스템의 보안을 위해 승인된 인원만 접근하도록 하는 절차입니다.

- **회원가입 (Signup) 구현:** 프론트엔드에서 회원가입 폼을 제공하여 이름, ID(또는 이메일), 비밀번호, 연락처 등을 입력받고, 희망하는 역할(간사 또는 평가위원)을 선택하게 할 수 있습니다. 가입 요청은 공개 API인 `POST /api/register`로 보내집니다. 백엔드에서는 새로운 User 계정을 생성하되, **활성화 필드(`is_active`)를 false**로 설정하여 미승인 상태로 저장합니다 [7]. 또한 사용자 역할(`role`)도 함께 저장하되 관리자 권한으로는 가입할 수 없도록 제한합니다 (관리자는 기본적으로 시스템에 1명 존재하거나, DB 스크립트로만 생성). 예를 들어 **FastAPI**에서의 회원가입 처리 코드:

```
from passlib.context import CryptContext
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

@app.post("/api/register")
async def register_user(user: UserCreateSchema):
```

```
    # UserCreateSchema에는 username, password, role, email 등 포함
    # 1. 중복 사용자 검사
    if await db.users.find_one({"login_id": user.username}):
        raise HTTPException(status_code=400, detail="이미 존재하는 사용자 ID")
    # 2. 비밀번호 해시화 및 계정 생성
    hashed_pw = pwd_context.hash(user.password)
    new_user = {
        "login_id": user.username,
        "password_hash": hashed_pw,
        "user_name": user.fullname,
        "email": user.email,
        "role": user.role if user.role in ("secretary", "evaluator") else "evaluator",
        "created_at": datetime.utcnow(),
        "is_active": False    # 최초에는 비활성 (관리자 승인 필요)
    }
    await db.users.insert_one(new_user)
    return {"message": "등록 완료, 관리자 승인 후 이용 가능합니다."}
```

위 코드에서 새로 가입한 사용자는 `is_active: False`로 저장됨을 확인할 수 있습니다 [8] . 가입 성공시 프론트엔드에는 "승인 대기 중"이라는 메시지를 표시하고 로그인 시도를 막을 수 있습니다. 그러나 보안을 위해 **백엔드 로그인 로직에서도 미승인 계정은 인증을 불허**해야 합니다.

- **로그인 시 승인 상태 검증:** 인증 엔드포인트 (`POST /api/login`)에서 사용자가 입력한 자격증명 검증 후 계정의 `is_active` 값이 False일 경우, 토큰 발급을 거부하고 에러를 반환합니다. 예를 들어:

```
@app.post("/api/login")
async def login_user(data: LoginSchema):
    user = await db.users.find_one({"login_id": data.username})
    if not user or not pwd_context.verify(data.password, user["password_hash"]):
        raise HTTPException(status_code=401, detail="아이디 또는 비밀번호 오류")
    if not user.get("is_active", False):
        raise HTTPException(status_code=403, detail="관리자 승인 대기 중인 계정입니다.")
    # ...이하 JWT 생성 및 응답 (id, username, role 포함)...
```

이로써 승인되지 않은 사용자는 올바른 비밀번호를 입력해도 로그인을 완료할 수 없습니다. 프론트엔드에서도 이 오류 메시지를 받아 "현재 계정은 승인 대기 상태"를 알려줍니다.

- **관리자에 의한 승인:** 관리자는 별도의 **회원 승인 관리 화면**에서 현재 `is_active=False`인 **미승인 사용자 목록**을 조회하고 개별 승인/거절 처리를 할 수 있습니다. 구현을 위해 백엔드에 관리자 전용 API 엔드포인트를 제공합니다:
- `GET /api/users?active=false` : 미승인 사용자 목록 조회
- `PUT /api/users/{user_id}/approve` : 특정 사용자 승인 (활성화) 처리
- (선택) `DELETE /api/users/{user_id}` : 가입 거절 시 사용자 삭제 또는 `role: rejected` 등으로 표시

예를 들어 승인 API를 구현하면 다음과 같습니다 (`admin_required` 디펜던시 사용):

```
@app.put("/api/users/{user_id}/approve", dependencies=[Depends(admin_required)])
async def approve_user(user_id: str):
```

```
    result = await db.users.update_one({"_id": user_id}, {"$set": {"is_active": True}})
    if result.modified_count == 0:
        raise HTTPException(status_code=404, detail="해당 ID의 사용자가 없습니다")
    return {"message": "승인 완료"}
```

관리자가 프론트엔드에서 이 API를 호출하여 성공 응답을 받으면, 해당 사용자 계정은 `is_active: True`로 전환되어 이제 로그인 및 시스템 이용이 가능해집니다. **Role 기반 접근 제어**에 따라 이 엔드포인트는 관리자만 호출 가능하며, 간사는 호출 권한이 없습니다.

- **프론트엔드 관리자 UI:** 관리자를 위한 "회원 승인" 페이지에서는 가입 요청이 들어온 사용자들의 리스트를 보여줍니다 (예: 테이블 형태로 ID, 이름, 이메일, 신청 역할 등을 표시). 각 행에는 “승인” 및 “거절” 버튼을 배치하여, 관리자가 조치할 수 있게 합니다. 승인 버튼 클릭 시 앞서 언급한 `approve_user` API를 호출하고, 성공 시 해당 사용자를 리스트에서 제거하거나 상태를 업데이트합니다.

요약하면, **Online-evaluation 플랫폼**에서는 **React + FastAPI** 아키텍처를 기반으로 각 역할에 맞는 UI 컴포넌트와 REST API를 설계하여 위 기능들을 구현할 수 있습니다 [1]. 권한별 접근 제어를 철저히 적용하여 (1) 관리자/간사는 평가 생성 및 설정, 평가위원 배정, 회원 승인을 수행하고, (2) 평가위원은 배정된 평가에 한해서만 점수를 입력할 수 있도록 하였습니다 [2]. 이러한 모듈별 설계는 유지보수성과 보안을 높이며, 나아가 **실시간 대시보드**나 **결과 보고서 출력 (PDF/Excel)** 등의 부가 기능과도 연계될 수 있는 확장성을 갖추고 있습니다.

---

[1] [2] [4] [5] [6] README.md
https://github.com/bruce0817kr/Online-evaluation/blob/edca46cdc8cda77f44e260c30f1a97a788741ce1/README.md

[3] COMPREHENSIVE_FIX_PLAN.md
https://github.com/bruce0817kr/Online-evaluation/blob/edca46cdc8cda77f44e260c30f1a97a788741ce1/COMPREHENSIVE_FIX_PLAN.md

[7] insert_admin_user.js
https://github.com/bruce0817kr/Online-evaluation/blob/edca46cdc8cda77f44e260c30f1a97a788741ce1/insert_admin_user.js

[8] fix_user_creation.py
https://github.com/bruce0817kr/Online-evaluation/blob/edca46cdc8cda77f44e260c30f1a97a788741ce1/fix_user_creation.py