ChatGPT

QualiJournal 관리자 시스템 최종 가이드북

1. 최종 프로젝트 구조 개요

QualiJournal 관리자(Admin) 시스템의 전체 디렉토리 구성은 다음과 같으며, 각 폴더와 주요 모듈의 역할은 아래와 같습니다:

```
project-root/
├── server_quali.py            # FastAPI 기반 백엔드 서버 (Admin API 엔트리포인트)
├── index.html                 # 관리자 대시보드 프론트엔드 (단일 HTML 페이지)
├── config.json                # 시스템 설정 파일 (게이트 임계값, 기능 토글 등 설정값 저장)
├── tools/                     # 수동 스크립트 모음 (뉴스 데이터 처리 유틸리티)
│   ├── make_daily_report.py   # 일일 보고서 Markdown 생성 스크립트
│   ├── enrich_cards.py        # 뉴스 카드 요약 및 번역 스크립트
│   └── ... 기타 데이터 병합/보정 스크립트 (예: rebuild_kw_from_today.py 등)
├── data/                      # (선택) 데이터 파일 저장 폴더 (JSON 파일을 이곳에 둘 수도 있음)
│   ├── selected_keyword_articles.json   # 수집된 전체 기사 목록 JSON (하루 키워드 기준) [1] [2]
│   └── selected_articles.json           # 승인된 기사 목록 JSON (발행 대상 최종본) [3]
├── archive/                   # 발행 아카이브 폴더 (날짜별 결과물 저장)
│   ├── reports/               # 생성된 일일 보고서 Markdown/HTML 저장 경로 [4]
│   ├── enriched/              # 요약/번역 Markdown 파일 저장 경로
│   └── ... (과거 발행본 .json/.md/.html 파일들) [4]
├── tests/                     # 통합 테스트 스크립트 (Pytest 기반)
├── Dockerfile                 # Cloud Run 배포를 위한 도커 컨테이너 설정 파일
├── .env                       # 환경변수 설정 파일 (ADMIN_TOKEN, DB_URL 등 **Git에 커밋하지 않음**)
└── .github/workflows/ci.yml   # CI/CD 파이프라인 정의 (GitHub Actions를 통한 자동 배포 등)
```

- **주요 JSON 데이터 파일**: `selected_keyword_articles.json` 파일에는 하루 키워드에 대해 수집된 모든 기사의 제목, 요약, 출처, 날짜, 점수, 승인여부 등이 담겨 있으며, 편집자는 이 파일에서 각 기사에 `approved` 플래그와 `editor_note`(코멘트)를 추가합니다 [1]. `selected_articles.json` 파일에는 그 중 **승인된 기사만**을 모아 저장하며 최종 발행 시 사용됩니다 [3]. 이 두 파일은 기본적으로 프로젝트 루트나 `data/` 폴더에 위치하지만, 백엔드 로직에서 경로를 유연하게 탐색하므로 실제 위치에 관계없이 불러올 수 있습니다 [5] [6].

- **tools 폴더**: 뉴스 수집 및 발행 과정에서 사용하는 여러 유틸리티 스크립트를 포함합니다. 예를 들어, `make_daily_report.py`는 현재까지 승인된 기사 목록(`selected_articles.json`)을 모아 **일일 보고서** (Markdown 형식)를 생성하는 스크립트이고, `enrich_cards.py`는 수집된 기사들에 대한 **요약 및 번역**을 수행하여 JSON 필드에 반영해주는 스크립트입니다. 그 외에도 `rebuild_kw_from_today.py` (수집된 원본 JSON들을 표준 구조로 병합) [7], `augment_from_official_pool.py` (승인된 기사 수가 부족할 때 공식 뉴스풀에서 기사 보충) [8], `force_approve_top20.py` (점수 상위 20개 기사 일괄 승

인) [9], `sync_selected_for_publish.py`와 `repair_selection_files.py` (승인된 기사만 발행 JSON으로 동기화하고 구조를 교정) [10] 등의 보조 스크립트가 포함되어 있습니다. 이러한 스크립트들은 **Windows 환경**에서는 PowerShell 스크립트(`run_quali_today.ps1`)로 순차 실행되기도 하며 [11], 백엔드 API에서도 필요 시 `_run_py` 헬퍼를 통해 호출됩니다 [5].

- **archive 폴더**: 매일 발행된 결과물을 보존하기 위한 경로입니다. 예를 들어, 2025년 10월 11일에 `AI` 키워드로 발행된 뉴스는 `archive/reports/2025-10-11_AI_report.md` 및 `.html`, 그리고 JSON 원본(만약 저장한다면) `2025-10-11_AI.json` 형태로 저장됩니다 [4]. 하루에 한 번 발행하는 구조이므로 기본적으로 날짜별로 한 세트의 파일이 생성되며, **중복 발행 시**에는 파일명이 겹치지 않도록 키워드 뒤에 타임스탬프(예: `_HHMM`)를 붙이는 등의 규칙을 마련해야 합니다 [12]. `archive/enriched` 폴더에는 해당 키워드의 **요약/번역본 Markdown 파일**이 저장됩니다 (예: `YYYY-MM-DD_<키워드>_ALL.md` 및 `_SELECTED.md`) [13] [14]. 이 archive 폴더는 시간이 지남에 따라 용량이 늘어나므로 정기적인 백업 및 관리가 필요합니다 [12].

- **백엔드 서버 코드**: `server_quali.py`는 FastAPI 기반의 서버로, 관리자 API 엔드포인트들을 정의하고 위 JSON 데이터 및 스크립트를 활용하여 **보고서 생성**, **요약/번역**, **내보내기**, **상태 조회** 등의 기능을 제공합니다. 이 서버는 Config 파일 및 환경변수를 불러와 초기화되며, 토큰 인증 미들웨어와 CORS 설정 등이 적용되어 있습니다 (자세한 내용은 뒷부분 보안 섹션 참조). 서버 시작 시에는 `config.json` 설정을 로드하고, 필요한 경우 `archive/`나 `data/` 경로에 있는 JSON 데이터를 미리 읽어 캐싱하기도 합니다 [15]. Config 파일 경로도 유연하게 탐색하도록 구현되어, 실제 위치와 상관없이 `config.json`을 찾아 사용합니다 [16].

- **프론트엔드 (index.html)**: 관리자 UI는 단일 HTML 파일 (`index.html`)로 구성되어 있으며, JavaScript를 통해 백엔드 API와 상호작용합니다. 이 UI에서는 **대시보드 형태**로 주요 KPI(전체 수집 기사 수, 승인된 기사 수 등)를 표시하고, **각종 작업 버튼** (예: "일일 보고서 생성", "키워드 요약/번역", "선정본 요약/번역", "Export MD", "Export CSV")을 제공합니다. 사용자가 버튼을 클릭하면 JavaScript가 해당 API 엔드포인트를 호출하여 작업을 수행하고, 결과를 화면에 표시하거나 파일 다운로드를 트리거합니다 [17] [18]. UI는 응답 상태에 따라 로딩 메시지, 성공 시 결과 링크 표시, 실패 시 오류 메시지 표시 등의 처리를 수행하며, 자세한 내용은 후술하는 "프론트엔드 UI 연동" 섹션에서 다룹니다.

2. 백엔드 API 명세

QualiJournal Admin 시스템 백엔드(FASTAPI 기반)는 다음과 같은 핵심 API 엔드포인트들을 제공합니다. 각 엔드포인트는 **ADMIN 토큰 인증**이 요구되며 (자세한 내용은 아래 '인증 미들웨어' 참조), JSON 데이터를 주고받거나 파일 다운로드를 처리합니다.

`/api/report` (POST) – 일일 보고서 생성

- **기능**: 편집자가 승인한 기사들로 구성된 **데일리 리포트**(마크다운)를 생성합니다.
- **동작**: 호출 시 내부적으로 `tools/make_daily_report.py` 스크립트를 실행하여 현재까지 승인된 기사 목록(`selected_articles.json`)을 취합합니다 [19]. 그 결과로 Markdown 형식의 일일 보고서 파일을 생성하며, 파일명은 `archive/reports/YYYY-MM-DD_report.md` 형식을 따릅니다 (키워드가 있다면 파일명에 슬러그로 포함됨: `YYYY-MM-DD_<키워드>_report.md`) [20]. 스크립트 실행 중 오류가 발생하더라도 **오류와 무관하게 항상 보고서 파일을 최종 생성**하도록 구현되었습니다 [20]. 즉, `make_daily_report.py`에서 예외가 발생하거나 일부 기사 데이터에 문제가 있더라도, 백엔드가 이미 수집된 JSON 데이터를 직접 읽어 최소한의 Markdown 보고서를 만들어내므로 항상 보고서를 얻을 수 있습니다.
- **요청 본문**: 필요 없습니다. (별도의 파라미터 없이도 현재 day's 데이터를 기반으로 동작)
- **응답**: 보고서 생성 결과를 나타내는 JSON을 반환합니다. 예시:

```
{
 "ok": true,
 "path": "archive/reports/2025-10-11_AI_report.md"
}
```

`ok` 필드는 보고서 생성 성공 여부를 나타내며, `path`는 생성된 Markdown 파일의 경로입니다. (실패 시에도 HTTP 200으로 응답하되 `ok: false`와 오류 메시지 등이 포함될 수 있습니다.) 생성된 Markdown 파일에는 편집자가 승인한 기사들의 **제목, 원문 링크, 요약, 편집자 코멘트** 등이 템플릿에 따라 정리되어 들어갑니다 [21].

`/api/enrich/keyword` (POST) – 키워드 전체 뉴스 요약/번역

- **기능**: **금일 키워드의 모든 수집 기사**에 대한 요약 및 번역 작업을 수행하고, 결과를 Markdown 파일로 저장합니다.
- **동작**: 호출 시 내부적으로 `tools/enrich_cards.py` 스크립트를 실행하여, `selected_keyword_articles.json` 내 모든 기사에 대해 **요약문(영문)**을 생성하고 (필요한 경우 번역 API 키가 설정되어 있으면 **한국어 번역**도 함께 수행) JSON 데이터에 반영합니다 [22]. 그런 다음, 스크립트 실행 성공 여부와 상관없이 백엔드 서버는 해당 JSON의 내용을 읽어 각 기사 카드의 **제목, 원문 링크, 영문 요약, 국문 요약, 편집자 코멘트**를 포함한 Markdown 파일을 만듭니다 [23] [24]. 생성되는 파일명은 `archive/enriched/YYYY-MM-DD_<키워드>_ALL.md` 형식이며, 해당 키워드의 전체 기사 요약본을 담고 있습니다 [13]. 스크립트가 없거나 실행 중 오류가 발생해도, 백엔드에서는 이미 수집된 JSON 데이터의 필드를 이용하여 **idempotent**하게 Markdown 파일을 작성하도록 구현되었습니다 [13] [23].
- **요청 본문**: 필요 없음. (현재 선택된 키워드의 데이터를 자동 사용)
- **응답**: JSON으로 생성된 요약 파일의 경로를 반환합니다. 예:

```
{
 "ok": true,
 "path": "archive/enriched/2025-10-11_AI_ALL.md"
}
```

`ok`는 성공 여부, `path`는 생성된 요약 Markdown 파일 경로입니다. Markdown 파일에는 각 기사별로 **제목, 링크, 요약문(영문/국문), 코멘트**가 목록 형태로 포함됩니다 [23]. 만약 스크립트 실행이 실패했다면 `ok: false`와 함께 오류 메시지가 응답 JSON에 포함될 수 있지만, 그래도 Markdown 파일은 부분적으로라도 생성되어 있을 것입니다.

`/api/enrich/selection` (POST) – 선정본(승인 기사들) 요약/번역

- **기능**: 편집자가 **승인한 기사들만 추려서** 요약 및 번역을 수행하고, 결과를 Markdown으로 저장합니다.
- **동작**: 동작 과정은 `/api/enrich/keyword`와 유사하나, 대상이 전체가 아닌 승인된 기사 목록입니다. 내부적으로 `enrich_cards.py`를 실행하여 `selected_articles.json` 상의 기사들에 대한 요약/번역 필드를 채운 후, 해당 데이터를 이용해 **Markdown 요약본**을 작성합니다 [13] [14]. 파일명은 `archive/enriched/YYYY-MM-DD_<키워드>_SELECTED.md` 형식을 가지며, 금일 최종 선정본 기사들의 요약/번역 결과를 담고 있습니다 [25] [14]. 이 역시 스크립트 오류와 무관하게 Markdown이 생성되도록 설계되어 있으며, 응답에 경로를 반환합니다.
- **요청 본문**: 필요 없음.
- **응답**: `/api/enrich/keyword`와 동일한 구조의 JSON 응답을 반환합니다. 예:

```
{
 "ok": true,
 "path": "archive/enriched/2025-10-11_AI_SELECTED.md"
}
```


생성된 Markdown 파일 구조와 필드는 키워드 전체 요약과 유사하나, **승인된 기사들만 포함**된다는 점이 다릅니다. 프론트엔드에서는 이 경로를 이용해 파일을 열거나 다운로드할 수 있습니다 [26].

`/api/export/md` (GET) – 최종 발행본 Markdown 내보내기

- **기능**: **최종 선정된 기사 리스트**(final selection)를 Markdown 파일 형태로 다운로드 받습니다.
- **동작**: 호출 시 현재 `selected_articles.json`에 포함된 **최종 기사 목록**을 Markdown 포맷으로 변환하여 즉시 응답으로 제공합니다 [21]. 이 Markdown은 편집된 기사들의 **제목, 요약, 원문 링크, 편집자 코멘트** 등을 모두 포함하는 템플릿으로 구성됩니다 [27]. 사실상 `/api/report` 엔드포인트가 생성하는 일일 보고서와 유사한 내용이 될 수 있지만, `/api/export/md`는 **파일 저장 없이 바로 다운로드용 콘텐츠를 응답**하는 데 초점이 있습니다.
- **요청 파라미터**: 별도 없음.
- **응답**: 응답 본문을 Markdown 텍스트로 반환하며, `Content-Disposition` 헤더를 통해 브라우저 다운로드를 트리거할 수도 있습니다 (예: `attachment; filename="final_selection.md"` 형태). 이 API는 프론트엔드에서 `<a href="/api/export/md" download>`와 같이 링크만 클릭해도 바로 파일로 저장되도록 사용할 수 있습니다 [28]. **Excel과의 호환성** 문제가 없는 형식이므로 (Markdown은 일반 텍스트), 별도 BOM(Byte Order Mark) 처리는 필요하지 않습니다.

`/api/export/csv` (GET) – 최종 발행본 CSV 내보내기

- **기능**: 최종 선정 기사 목록을 CSV 파일로 다운로드 받습니다.
- **동작**: 현재 `selected_articles.json`의 데이터를 불러와 각 기사의 주요 필드를 CSV (Comma-Separated Values) 형식으로 변환하여 응답합니다 [21]. CSV 컬럼에는 일반적으로 **기사 제목, 요약, 원문 URL, 편집자 코멘트, 기타 메타정보** 등이 포함될 수 있습니다 [29]. 이 CSV 출력은 편집자가 엑셀 등으로 열어서 분석하거나 별도 리포트 작성에 활용할 수 있도록 인코딩 등에 유의하여 구현되었습니다. 특히 **Excel에서 깨지지 않도록 UTF-8 BOM(Byte Order Mark)을 파일 맨 앞에 추가**하여 내보냅니다 [30] [29].
- **요청 파라미터**: 없음.
- **응답**: 응답 본문을 CSV 텍스트로 반환하며, 다운로드를 위해 `Content-Type: text/csv`와 `Content-Disposition: attachment; filename="final_selection.csv"` 헤더를 포함합니다. 프론트엔드에서는 이 엔드포인트를 호출하여 곧바로 CSV 파일 저장을 유도할 수 있습니다 (예: `<a href="/api/export/csv" download>` 형태) [28]. Windows 환경의 Excel에서도 한글이 깨지지 않도록 BOM이 포함된 점을 확인했습니다.

인증 토큰 미들웨어 – ADMIN Token 검증

- **개요**: 상기 모든 API 엔드포인트는 **관리자 전용** 기능이므로, 보안을 위해 **인증 토큰(Authentication Token)**을 요구합니다. 이를 위해 FastAPI 애플리케이션 수준에서 **커스텀 미들웨어** 또는 Dependency를 사용하여 요청 시 제공된 토큰이 유효한지 검증합니다.
- **설정**: 인증에 사용되는 토큰 값은 운영 시 **환경변수**(`ADMIN_TOKEN`)로부터 로드되며, 서버 기동 시 메모리에 저장됩니다 [31]. 이 토큰은 사전에 충분히 길고 예측 불가능한 값으로 설정해야 하며, .env 파일이나 Cloud Run의 시크릿 변수로 관리합니다.
- **사용 방법**: 클라이언트(프론트엔드)에서는 **모든 API 호출 시** 헤더에 `Authorization: Bearer <ADMIN_TOKEN>`을 포함해야 합니다. 서버는 요청 헤더의 토큰과 사전 설정된 토큰을 비교하여 다를 경우 401 Unauthorized 또는 403 Forbidden 에러를 반환합니다 [32]. 예를 들어, `/api/report`나 `/api/`

`enrich/...` 엔드포인트에 토큰 없이 호출을 시도하면 **즉시 401/403 에러**가 발생하는지를 확인하였고, 올바른 토큰을 포함하면 정상적으로 요청이 처리됨을 테스트했습니다 [32].

- **미들웨어 구현**: 구현 방식에는 두 가지 가능성이 있습니다. (1) **Starlette Middleware**로 모든 경로에 대해 요청을 가로채 헤더 검사 후 인증하거나, (2) **FastAPI Dependency**로 각 라우터/엔드포인트에 의존성을 주입하여 토큰을 검사하는 방법입니다. 본 시스템에서는 간단히 정적 토큰을 사용하는 점을 고려해, 경량화된 미들웨어 또는 데코레이터를 통해 모든 Admin API 경로에 **단일 토큰 체크**를 적용했습니다. 인증 실패 시 JSON 형태로 에러 메시지를 반환하며, 성공 시 다음 단계로 진행합니다.
- **CORS 및 HTTPS**: Token 인증과 함께, **CORS 정책**도 설정되어 있습니다. 관리자 웹 UI가 동작하는 도메인을 허용(origin whitelist)에 추가하고 그 외의 요청은 차단하도록 하여, 토큰이 유출되지 않도록 이중 보호합니다 [33]. Cloud Run 배포 환경에서는 기본적으로 HTTPS 통신이 이루어지므로, 클라이언트-서버 간 트래픽도 암호화된 상태로 전달됩니다 [34]. 개발 중에도 필요 시 로컬에 HTTPS를 적용하거나, 최소한 CORS 정책으로 특정 도메인(예: localhost 또는 사내망 도메인)만 허용하도록 설정하여 보안을 유지합니다 [35].

기타 API – 상태 조회 및 설정 변경

위 주요 기능 외에 관리자 대시보드에서 활용되는 추가 API가 있습니다:

- `/api/status` **(GET)**: 시스템의 **현황 및 KPI 지표**를 조회합니다. 이 엔드포인트는 수집/승인된 기사 수, 커뮤니티/공식 소스별 집계, 게이트 임계값과 현재 승인된 기사 수 비교 결과 등을 반환합니다 [36]. 예를 들어 `selection_total` (수집된 총 기사 수), `selection_approved` (승인된 기사 수), 각 상태별(article state) 카운트(candidate/ready/rejected), `community_total`, `keyword_total`, 그리고 현재 설정된 `gate_required` 값과 이를 충족했는지 여부(`gate_pass`) 등이 포함됩니다 [36]. 이 API는 관리자 UI가 주기적으로 호출하여 대시보드의 KPI 숫자들을 자동 갱신하는 데 사용됩니다 (예: 새로 기사가 승인되거나 임계값이 변경되면 즉각 업데이트) [37].

- `/api/config/gate_required` **(PATCH)**: **게이트 임계값**(최소 필요 승인 기사 수)을 조정하기 위한 엔드포인트입니다. 요청 바디로 `{"value": <number>}` 형태의 JSON을 받으면, 서버 메모리에 유지 중인 게이트 임계값을 업데이트함과 동시에 `config.json` 파일의 `features.gate_required` 설정값도 변경하여 **영구적으로 저장**합니다 [15]. 또한 `external_rss.gate_required` 등 관련 설정도 함께 갱신하여 설정 파일의 일관성을 유지합니다 [38]. 이 엔드포인트를 통해 운영자는 웹 UI상의 슬라이더를 움직여 임계값을 높이거나 낮출 수 있고, 변경 즉시 `/api/status`의 `gate_required`와 `gate_pass` 값이 반영되어 UI에 나타납니다 [39] [37]. Cloud Run 같은 컨테이너 환경에서는 인스턴스 재시작시 메모리 값이 초기화될 수 있으므로, 이렇게 **파일에 영구 저장**하는 방식으로 설정 변경을 안전하게 유지합니다 [40]. (단, 멀티 인스턴스 환경에서는 각 인스턴스가 별도 파일을 갖고 있으므로, 임계값 변경 시 모든 인스턴스에 적용하려면 공용 DB 또는 Config 중앙관리 방식이 향후 필요할 수 있습니다.)

이상 주요 백엔드 API들과 인증 구조를 살펴보았습니다. 다음 섹션에서는 이러한 API들을 프론트엔드(UI)와 연동하여 실제 운영자가 어떻게 사용할 수 있는지 알아보겠습니다.

3. 프론트엔드 UI 연동 방법

관리자 웹 UI(`index.html`)는 백엔드 Admin API와 상호작용하여 보고서 생성, 뉴스 카드 요약/번역, 결과물 다운로드 등의 기능을 제공합니다. UI 코드는 **Vanilla JS**로 작성되었으며, 각 버튼에 대한 이벤트 핸들러를 통해 API 호출 -> 응답 처리 -> 결과 표시 순으로 동작합니다 [18]. 아래에서는 핵심 기능별로 **index.html 수정 및 연동 예시**를 설명합니다:

3.1 보고서 생성 버튼 (`일일 보고서 생성`)

- **버튼 추가**: 대시보드 HTML에 `<button id="btnReport">일일 보고서 생성</button>` 요소를 추가합니다. 이 버튼은 클릭 시 `genReport()` 함수를 호출하도록 onclick 핸들러를 지정합니다. 버튼 옆에는 결과를 열람할 수 있는 링크 `<a id="linkReport" href="#" target="_blank" style="display:none;">보고서 열기</a>`를 미리 배치해 둡니다 [18]. 처음에는 `display: none`으로 숨겨두었다가, 보고서 생성 성공 시 링크를 표시하여 사용자가 생성된 Markdown을 열 수 있게 합니다.

- **JS 함수 구현**: `genReport()` 함수에서는 다음 순서로 동작합니다:

- **로딩 상태 표시**: 보고서 생성은 수 초가 걸릴 수 있으므로, 버튼 클릭 시 우선 `btnReport.disabled = true`로 버튼을 비활성화하고 사용자에게 "보고서 생성 중..."과 같은 로딩 메시지를 표시합니다 [41]. 필요하다면 버튼 또는 화면 내 특정 영역에 스피너나 진행 중임을 나타내는 요소를 노출합니다.
- **API 호출**: `fetch('/api/report', { method: 'POST', headers: { 'Authorization': 'Bearer <토큰>' } })` 형식으로 Admin API를 호출합니다. 이때 토큰이 필요한 경우 헤더에 포함해야 하며 (토큰 값은 안전을 위해 UI에 하드코딩하지 않고, 운영자가 직접 입력하거나 별도의 설정을 통해 주입하는 방식을 고려해야 함), POST 메소드로 요청을 보냅니다.
- **응답 처리**: 응답이 성공하면 (`res.ok`가 true) 반환된 JSON을 파싱하여 `data.path`를 얻습니다. 콘솔 로그 영역에 전체 JSON 응답을 출력하여 디버깅에 참고할 수 있게 하고 [41], 앞서 숨겨둔 링크 `linkReport`의 href를 해당 경로로 설정하고 링크를 화면에 표시합니다 (e.g., `linkReport.style.display = 'inline'`).
- **오류 처리**: 만약 응답 상태가 200이 아니거나 `data.ok`가 false인 경우, 오류로 간주합니다. 이때 화면의 로그/메시지 영역에 `console.error`를 사용해 `data` 또는 `err` 내용을 출력하고, 사용자에게 "보고서 생성 실패: ... 이유 ..." 등의 메시지를 표시합니다 [41]. (예: 로그 박스에 서버에서 전달한 error 메시지를 보여주거나, `alert()`을 사용할 수도 있습니다.)

- **상태 복원**: 작업 완료 후 (성공 또는 실패 여부와 무관하게) 버튼을 다시 활성화하기 위해 `btnReport.disabled = false`로 설정합니다. 또한 로딩 메시지가 있었다면 이를 지웁니다.

- **index.html 수정 예시**: 아래는 간략한 예시 코드입니다:

```
<button id="btnReport" onclick="genReport()">일일 보고서 생성</button>
<a id="linkReport" href="#" target="_blank" style="display:none;">보고서 열기</a>
<script>
 async function genReport() {
  const btn = document.getElementById('btnReport');
  const link = document.getElementById('linkReport');
  btn.disabled = true;
  link.style.display = 'none';
  // (로딩 메시지 표시 로직 추가 가능)
  try {
   const res = await fetch('/api/report', { method: 'POST', /*headers...*/ });
   const data = await res.json();
   console.log(data);  // 응답 JSON을 로그에 출력
   if (res.ok && data.path) {
    link.href = '/' + data.path;     // 경로 설정 (서버 정적파일 경로 기준)
    link.style.display = 'inline';   // 링크 표시
   } else {
    console.error('Report generation failed:', data);
```

```
      alert('보고서 생성 실패: ' + (data.error || '원인 불명'));
    }
   } catch (err) {
    console.error('Error:', err);
    alert('보고서 생성 중 오류 발생');
   } finally {
    btn.disabled = false;
   }
  }
</script>
```

위 코드는 개략적인 흐름을 나타낸 것이며, 실제 구현에서는 토큰 인증이 필요한 경우 `fetch`에 헤더를 추가해야 합니다. 또한 응답받은 `path`는 서버 내 경로이므로, 정적 파일 제공 설정에 따라 `/archive/...`로 접근하거나 별도의 API로 파일 내용을 불러와 모달로 표시하는 등의 UI 확장이 가능합니다 [17].

3.2 뉴스 카드 요약/번역 버튼 (키워드 전체 & 선정본)

- **버튼 구성**: 키워드 전체 기사 요약을 위한 버튼 `<button id="btnEnrichAll">키워드 요약/번역</button>`과, 최종 선정본만 요약하기 위한 `<button id="btnEnrichSel">선정본 요약/번역</button>` 버튼을 UI에 추가합니다. 각 버튼의 onclick으로 각각 `enrichKeyword()`와 `enrichSelection()` 함수를 호출하도록 설정합니다. 그리고 결과 파일을 열 수 있는 링크 `<a id="linkEnrichAll">...ALL.md 열기</a>`, `<a id="linkEnrichSel">...SELECTED.md 열기</a>`를 미리 준비하여 숨겨둡니다 [18] [26].

- **공통 동작**: 두 함수 `enrichKeyword()`와 `enrichSelection()`의 로직은 보고서 생성과 매우 유사합니다. 차이점은 호출하는 API 엔드포인트가 `/api/enrich/keyword` 또는 `/api/enrich/selection`라는 것뿐입니다. 구현 단계는 다음과 같습니다:

- 버튼 비활성화 및 로딩 표시 (`disabled=true`, 메시지 출력 등).
- 각 API로 POST 요청 (`fetch('/api/enrich/keyword', {...})` 등) 보내기.
- 응답 JSON에서 `path` 추출하여 링크 href 설정, 링크 표시 [26].
- 실패 시 로그 출력 및 사용자 알림.

- 마무리로 버튼 활성화 복구.

- **UI 처리 특징**: 요약/번역 결과물은 Markdown 파일로 생성되는데, **내용이 번역된 기사 요약문들**이므로 UI에서 바로 열어볼 수 있도록 하는 것이 좋습니다 [42]. 따라서 링크 클릭 시 새 탭으로 해당 Markdown을 열리게 하거나, 필요하다면 파일을 다운받도록 `download` 속성을 줄 수도 있습니다. 또는 UI 상에서 Markdown 내용을 AJAX로 불러와 모달 창이나 별도의 DIV에 렌더링하는 방식도 고려 가능합니다 (다만 초기 단계에서는 링크를 통해 원본 `.md` 파일을 열람하는 정도로 충분할 것입니다).

- **에러 처리**: 만약 요약 파일 생성 API가 실패할 경우 (예: `ok: false`), 응답에 포함된 오류 메시지를 로그에 출력하고 UI에도 표시해야 합니다 [24]. 예를 들어, 번역 API 키가 없어서 `enrich_cards.py`가 제대로 동작하지 않았다면, 응답 JSON에 `"error": "Translation API key not set"` 같은 메시지가 담길 수 있으므로 이를 보여주어 **운영자가 원인을 파악**할 수 있게 합니다. 현재 구현에서는 실패 시 **기존 JSON 응답을 통째로 로그 영역에 출력**하도록 되어 있어 디버깅에 도움이 됩니다 [43].

3.3 결과물 Export 버튼 (MD, CSV 내보내기)

- **버튼 추가**: UI에 최종 결과를 파일로 저장할 수 있는 "Export MD", "Export CSV" 버튼 (또는 드롭다운 메뉴)을 추가합니다 [44]. 여기서는 예시로 각각의 버튼을 둔다고 가정합니다:

```
<a id="btnExportMd" href="/api/export/md" download="QualiNews_Today.md">Export MD</a>
<a id="btnExportCsv" href="/api/export/csv" download="QualiNews_Today.csv">Export CSV</a>
```

위와 같이 `<a>` 태그를 사용하고 `href`를 해당 API 엔드포인트로 지정하며 `download` 속성에 기본 파일 이름을 넣어 두면, 사용자가 버튼(링크)을 클릭할 때 즉시 파일 다운로드가 이루어집니다 [28]. (※ `<button>`으로 구현하는 경우 JavaScript로 fetch 호출 후 Blob 처리하여 다운로드를 트리거하는 방법도 있으나, 단순 링크로도 처리가 가능하므로 편의상 `<a>`를 사용합니다.)

- **동작 확인**: 이 방식으로 구현하면 별도의 JS 코드 작성 없이도, 브라우저가 GET 요청을 보내고 응답을 파일로 저장합니다. 단, 이러한 `<a>` 링크에도 인증 토큰이 필요할 수 있습니다. 만약 토큰 인증 미들웨어가 활성화되어 있다면, 단순 클릭으로는 헤더에 토큰이 전달되지 않으므로 **인증이 실패**합니다. 이를 해결하는 방법 중 하나는 **토큰을 URL 파라미터**로 첨부하는 것입니다 (예: `/api/export/md?token=...` 형태). 그러나 이는 토큰 노출 우려가 있어 권장되지 않습니다. 다른 방법으로는, **사용자가 Export 버튼을 클릭하면 JS 함수에서 `fetch`로 파일 데이터를 받아 처리**하는 것입니다. 예를 들어:

```
async function exportCsv() {
 const res = await fetch('/api/export/csv', { headers: { Authorization: 'Bearer ...'} });
 if (res.ok) {
  const blob = await res.blob();
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = 'QualiNews_Today.csv';
  document.body.appendChild(a);
  a.click();
  a.remove();
 } else {
  alert('CSV export failed');
 }
}
```

이런 식으로 토큰을 포함해 데이터를 받아 브라우저 다운로드를 트리거할 수 있습니다. 구현 난이도에 따라, **동일 도메인에서 UI와 API가 운영**되고, 내부 사용자만 쓴다면 일시적으로 토큰 인증을 Export 엔드포인트에 한해 비활성화하는 방안도 검토할 수 있습니다. 그러나 원칙적으로는 모든 API에 인증을 거는 것이 안전합니다.

- **Excel 인코딩 주의**: CSV 다운로드의 경우, 실제로 파일을 열어 **한글이 깨지지 않는지** 확인해야 합니다. 앞서 백엔드에서 BOM을 추가했으므로 Windows Excel에서 바로 열면 UTF-8이 올바르게 인식되어 표시됩니다 [29]. 운영자는 Export CSV 파일을 정기적으로 받아 **백업**하거나 **통계 분석**에 활용할 수 있습니다.

3.4 로딩 상태 및 에러 처리 예시 (UI 공통 UX)

위에서 각 기능별로 일부 다뤘지만, 프론트엔드 UI 구현 시 전반적으로 고려해야 할 UX 패턴은 다음과 같습니다:

- **버튼 중복 클릭 방지**: 하나의 작업이 진행 중일 때 사용자가 여러 번 버튼을 눌러 중복 요청을 보내지 않도록, API 호출 시작 시 버튼을 비활성화하고 완료 후 다시 활성화합니다 [41]. 예를 들어, "보고서 생성" 진행 중에는 해당 버튼을 회색으로 비활성화하고 커서를 `wait`로 바꾸는 등의 시각적 표시를 해줍니다.

- **로딩 피드백**: 작업 실행 후 결과가 나오기까지 일정 시간(몇 초 이상)이 소요될 경우, 사용자가 상태를 알 수 있도록 "로딩중..."과 같은 문구나 스피너 GIF 등을 표시합니다. index.html에는 로그 출력 영역이 있다면 그곳에 temporarily "Generating report..." 메시지를 넣었다가 완료 시 교체하거나 지우는 식으로 구현할 수 있습니다.

- **성공 시 UI 업데이트**: API 호출이 성공적으로 끝나면, 해당 작업의 결과물에 접근할 수 있는 UI 요소를 노출합니다. 앞서 설명한 대로 링크 태그를 미리 만들어두고 `display:none` 처리했다가, 성공 시 `display:block/inline`으로 전환하는 방식이 간단합니다 [18]. 예컨대 "보고서 열기" 링크, "요약본 열기" 링크 등이 그러합니다. 추가로, `/api/status` 등을 다시 호출하여 KPI 숫자를 갱신하거나, 새로운 데이터에 맞춰 화면 일부를 리프레시하는 것도 고려할 수 있습니다.

- **실패 시 알림**: API 응답이 오류를 반환하거나 네트워크 실패가 발생한 경우, 사용자가 이를 인지하고 조치할 수 있도록 명확히 알려야 합니다. 콘솔에만 오류를 찍어 놓으면 실제 운영자는 알기 어렵기 때문에, **화면상의 로그 영역에 빨간색 오류 메시지**를 출력하거나 `alert()` 팝업을 띄워 "작업에 실패했습니다: 원인 ..."을 보여줍니다 [41]. 예를 들어 토큰 인증 실패로 401이 온 경우 "인증 오류: 관리 토큰을 확인하세요"와 같이 안내할 수 있고, 요약/번역 기능에서 API 키 누락으로 실패한 경우 "요약 생성 실패: 번역 API 키 설정 필요" 등의 구체적인 원인을 전달해주는 것이 좋습니다. (백엔드에서 이런 원인을 `error` 필드로 주는 경우 이를 활용합니다.)

- **예외 상황 처리**: 프론트엔드에서는 예기치 못한 예외(예: JS 오류, fetch exception 등)가 발생할 수 있으므로 `try...catch`를 통해 오류를 잡고 로그를 출력하도록 구현합니다. 그리고 필요시 버튼 비활성화 상태를 복구하는 `finally` 블록을 항상 넣어 두어야 합니다. 이는 위 예시 코드에서도 강조된 부분입니다.

정리하면, 프론트엔드 index.html은 **버튼 -> API 호출 -> 응답에 따라 UI 업데이트**라는 패턴으로 구성되며, 위 예시들을 참고하여 각 기능을 연동하면 됩니다. 실제 index.html 개선사항으로 위 기능들이 모두 반영되었으며, MD/CSV 내보내기 기존 기능도 새로운 링크 구조와 충돌하지 않고 그대로 동작하도록 유지되었습니다 [41].

4. Cloud Run 배포 및 CI/CD 자동화

QualiJournal Admin 서버는 **Docker 컨테이너**로 패키징되어 Google Cloud Run에 배포됩니다 [45]. 기존에는 수동으로 도커 이미지를 빌드하고 배포했으나, 최종 마무리 단계에서는 **CI/CD 파이프라인**을 구축하여 코드 변경 시 자동으로 배포가 이뤄지도록 개선되었습니다 [46]. 여기서는 Cloud Run 배포 절차와 GitHub Actions를 이용한 CI/CD 구성 방법을 안내합니다.

4.1 Docker 컨테이너 구성

- **Dockerfile 작성**: 프로젝트 루트에 Dockerfile을 배치합니다. 이 Dockerfile에서는 FastAPI 서버를 실행하기 위한 환경을 설정합니다. 예를 들어:

```
FROM python:3.10-slim
WORKDIR /app
COPY . /app
```

```
RUN pip install -r requirements.txt
CMD ["uvicorn", "server_quali:app", "--host=0.0.0.0", "--port=8080"]
```

위는 간략한 예시이며, 실제 구성에 따라 관리자 UI 정적 파일 제공을 위해 starlette StaticFiles 설정이나 gunicorn 사용 등을 추가할 수 있습니다. 중요한 것은 Cloud Run이 기본 포트 8080을 사용하므로, uvicorn/gunicorn이 8080 포트로 동작하도록 CMD를 설정하는 점입니다.

- **컨테이너 이미지 빌드**: 로컬에서 `docker build -t asia-northeast3-docker.pkg.dev/<GCP_PROJECT>/qualijournal/admin:latest .` 등의 명령으로 이미지를 빌드하고, `docker push`로 Google Artifact Registry 등에 푸시할 수 있습니다. 그러나 이후 설명할 CI/CD에서 이 과정을 자동화할 것이므로, 개발 단계에서만 수동 빌드를 사용합니다.

4.2 Cloud Run 서비스 설정

- **Cloud Run 생성**: GCP 콘솔 또는 gcloud CLI를 통해 Cloud Run 서비스를 만듭니다. 컨테이너 이미지를 지정하고 메모리/CPU, 동시접속 등 매개변수를 설정합니다. QualiJournal Admin의 경우 **CPU 1개, 메모리 512MB~1GB** 정도로 시작할 수 있으며, 동시 요청 수에 따라 autoscaling 설정을 조정합니다 (기본 동시 80, 최소/최대 인스턴스 등).
- **환경 변수 등록**: Cloud Run의 Variables & Secrets 설정에서 `.env` 파일에 정의한 환경변수들을 입력합니다 [47]. 예를 들어 `ADMIN_TOKEN`, `DB_URL`, `API_KEY` 등을 키-값으로 추가합니다. (Secrets로 등록하면 더욱 안전하며, 권한을 가진 서비스 계정만 접근하게 할 수 있습니다.) 이 단계에서 환경변수들의 값이 정확히 입력되었는지, 불필요한 민감정보 (예: 로컬 경로 등)가 포함되지 않았는지 확인합니다 [48].
- **도메인 및 TLS**: Cloud Run 배포 시 기본적으로 `https://<service-id>-<hash>-<region>.run.app` 형태의 도메인이 제공됩니다. Custom domain을 매핑하지 않더라도 이 URL로 접근 가능하며, HTTPS가 자동 적용됩니다 [34]. 필요한 경우 사용자 도메인을 연결하고 TLS 인증서를 적용할 수 있습니다.

4.3 GitHub Actions를 이용한 CI/CD 파이프라인

- **목표**: 코드가 main 브랜치에 푸시될 때 자동으로 **도커 이미지 빌드 -> 테스트 -> Cloud Run 배포**까지 진행되도록 Workflow를 구성합니다 [46]. 이렇게 하면 개발자가 코드 수정 후 푸시만 하면 몇 분 내로 새로운 버전이 Cloud Run 서비스에 반영되어 수동 개입 없이 배포가 완료됩니다 [49].

- **GCP 인증 세팅**: 먼저 GitHub Actions에서 Google Cloud에 접근할 수 있도록 서비스 계정을 준비합니다. GCP IAM에서 Deploy to Cloud Run 권한을 가진 서비스 계정을 만들고 키(JSON)를 생성하여, GitHub 리포지토리의 Secrets에 `GCP_SA_KEY`로 등록합니다. 또한 프로젝트 ID, Cloud Run 서비스명, GCP_REGION 등의 값도 Secrets 혹은 workflow env로 지정합니다.

- **Workflow YAML 작성**: `.github/workflows/deploy.yml` 등에 아래와 같은 Workflow를 정의합니다 (단계 요약):

- **트리거**: `push` 이벤트 (특정 브랜치, 예: `main`).
- **이미지 빌드**: actions/checkout으로 소스 체크아웃 -> `google-github-actions/setup-gcloud`로 gcloud 인증 (서비스계정 키 사용) -> `gcloud builds submit` 또는 Docker build/push 실행. 여기서는 Artifact Registry에 이미지 푸시를 위해 `gcloud builds submit --tag asia.gcr.io/$PROJECT_ID/qualijournal-admin:$GITHUB_SHA` 등을 사용할 수 있습니다.
- **Cloud Run 배포**: Google에서 제공하는 Action인 `google-github-actions/deploy-cloudrun`을 활용해, 이미 빌드된 이미지 경로와 Cloud Run 서비스명, 프로젝트, 지역 등을 인자로 전달하여 새 리비전을 배포합니다 [50]. 예시:

```
- uses: google-github-actions/deploy-cloudrun@v1
  with:
    image: asia.gcr.io/${{ env.GCP_PROJECT }}/qualijournal-admin:${{ github.sha }}
    service: qualijournal-admin
    project_id: ${{ env.GCP_PROJECT }}
    region: asia-northeast3
    env_vars: ADMIN_TOKEN=${{ secrets.ADMIN_TOKEN }},API_KEY=$
{{ secrets.API_KEY }},DB_URL=${{ secrets.DB_URL }}
```

`env_vars`를 통해 중요한 환경변수도 배포시에 세팅 가능합니다 (혹은 Cloud Run 콘솔에 미리 설정해두면 생략 가능).

- **테스트 단계 추가**: (선택) 배포 전에 Pytest를 실행하여 모든 테스트를 통과하는지 확인하고, 실패 시 배포를 중단하도록 구성합니다 [51] [52]. 이는 CI 신뢰성을 높이고 문제있는 코드가 배포되지 않게 해줍니다.

- **완료**: 배포가 성공하면 Actions 로그에 Cloud Run 서비스 URL을 출력하게 할 수 있습니다 [53]. 필요시 이 URL을 Slack 알림 등으로 전파하는 Step을 추가할 수도 있습니다.

- **CI/CD 동작 확인**: 설정 후 실제로 main 브랜치에 push하여 워크플로우가 잘 작동하는지 확인합니다. Actions 로그에서 **도커 이미지 빌드/푸시 성공 메시지**와 **Cloud Run 배포 성공 메시지**를 찾아보고, 마지막에 출력된 서비스 URL을 브라우저나 `curl`로 호출해 **응답이 오는지** 검사합니다 [54]. 특히 `/api/status` 엔드포인트를 호출해 서버 KPI 지표가 정상적으로 반환되는지 확인하는 것이 좋습니다 [55].

- **배포 후 검증 및 모니터링**:

- Cloud Run **환경변수** 탭에서 .env 내용이 제대로 반영되었는지 (예: ADMIN_TOKEN, API_KEY 값 등) 확인합니다 [56].
- Cloud Run **로그**를 실시간 모니터링하여, 새로운 컨테이너 시작 시 에러(traceback)나 Dependency 불일치, DB 마이그레이션 오류 등이 없는지 확인합니다 [57]. 오류 발생 시 해당 리비전을 **롤백**하거나, 패치를 통해 즉시 수정해야 합니다.

- 필요한 경우, `gcloud run services describe <서비스명>` 커맨드로 현재 배포된 이미지 태그, 환경변수, 트래픽 할당 등이 기대와 일치하는지 살펴봅니다 [58].

- **CI/CD 보안**: CI 파이프라인에 사용되는 GCP 서비스 계정 키, ADMIN_TOKEN 등은 GitHub Secrets에 저장되어 안전히 참조됩니다. Workflow 파일에 민감정보를 직접 넣지 않도록 주의해야 합니다. 또한, Cloud Run 배포시 Secrets Manager 연동을 사용하면 환경변수 대신 시크릿 참조로 주입할 수도 있습니다 (Cloud Run의 시크릿 기능 참고).

이와 같이 Cloud Run 자동 배포 환경을 구축하면, 개발부터 운영 배포까지 일관된 흐름으로 진행할 수 있습니다. 마지막으로, 배포 자동화 설정이 완료되면 **배포 결과를 체크리스트**를 통해 점검하여 제대로 동작함을 확인해야 합니다:

- **배포 후 체크포인트** [59]:
- Actions 로그에서 **이미지 빌드 및 Cloud Run 배포 성공**을 확인 (에러 없는지).
- **Cloud Run 서비스 URL 확인**: 배포된 URL로 접속하여 API 응답 확인. 예를 들어 `curl <서비스URL>/api/status`로 200 OK와 JSON 데이터를 수신 확인 [55].
- **환경변수 검증**: Cloud Run 콘솔의 Variables 섹션에서 필요한 변수들이 모두 설정되었는지 재확인 (ADMIN_TOKEN, DB_URL 등) [56].
- **로그 확인**: Cloud Run 로그에서 애플리케이션 시작 로그를 보고 오류 trace가 없는지, 필요한 초기화가 다 되었는지 확인. Dependencies 로드 실패나 이슈가 있다면 즉시 수정.

- **트래픽 라우팅**: (블루/그린 배포 등을 쓴 경우) 새로운 리비전에 트래픽 100% 할당이 되었는지 확인. `gcloud run services describe`로 확인 가능 [58].

CI/CD를 구성함으로써 운영자는 일상적으로 배포 작업에 드는 시간을 절약하고, 코드 변경 시 실수 없이 배포가 이뤄지는지 확인할 수 있습니다.

5. 환경 변수 및 보안 설정

운영 환경에서의 민감 정보와 설정 값들은 코드에 하드코딩하지 않고 **환경 변수**로 관리합니다. QualiJournal Admin 시스템에서 사용하는 주요 환경 변수와 보안 설정 방법은 다음과 같습니다:

- **ADMIN_TOKEN**: 관리자 API 액세스를 보호하는 인증 토큰 값입니다. `.env` 파일이나 Cloud Run 환경 변수로 설정하며, 충분히 복잡한 난수 문자열로 만듭니다 (예: `ADMIN_TOKEN=aklJE92nv2...`). 이 토큰은 FastAPI 서버가 기동 시 `os.getenv("ADMIN_TOKEN")`으로 불러와 전역 변수에 저장하고, **인증 미들웨어**에서 비교에 사용합니다 [31] [60]. 절대로 소스 레포지토리에 노출되지 않도록 하고, 필요한 경우 정기적으로 변경(rotate)할 수 있습니다. 토큰을 변경했다면 Cloud Run에 새로운 값으로 업데이트하고 서비스를 재시작해야 새로운 토큰이 적용됩니다.

- **DB_URL**: PostgreSQL 등 데이터베이스 연결 정보를 담은 URL입니다. 예시: `DB_URL=postgresql://username:password@hostname:5432/database_name`. 현재 시스템은 아직 JSON 파일 기반으로 동작하지만, 향후 **PostgreSQL 연동** 시 이 변수를 사용합니다 [61]. 서버 기동 시 `os.getenv("DB_URL")`로 값을 읽어, 만약 설정되어 있다면 ORM 또는 DB 커넥션을 초기화하는 로직이 추가될 예정입니다. (DB 연동 준비에 관한 내용은 다음 섹션 참고)

- **API_KEY**: 뉴스 요약/번역 기능에 사용되는 외부 API 키입니다. OpenAI GPT-4나 DeepL, 또는 Naver Papago 등 어떤 서비스를 쓰는지에 따라 키 이름이 다를 수 있으나, 예를 들어 `OPENAI_API_KEY` 또는 `PAPAGO_CLIENT_ID`, `PAPAGO_SECRET` 등의 변수가 있을 수 있습니다. 편의상 여기서는 `API_KEY`로 통칭합니다. 이 키가 `.env`에 설정돼 있으면 `enrich_cards.py` 스크립트가 해당 API를 호출하여 영문 요약문을 한국어로 번역하는 기능을 수행할 수 있습니다 [62]. 키가 없으면 번역은 건너뛰고 영문 요약만 사용합니다.

- **기타 설정 변수**:

- **GATE_REQUIRED**: 품질 게이트(최소 승인 필요 기사 수)를 환경변수로도 지정할 수 있습니다. Config 파일(`config.json`)의 `features.gate_required`를 서버가 읽지만, **동일한 설정이 환경변수로 주어지면 그 값을 우선시**하도록 구현되어 있습니다 [63]. 따라서 `GATE_REQUIRED=10` 같이 지정하면 기본 임계값이 10으로 적용됩니다 (물론 이후 슬라이더 조정으로 바뀔 수 있음).
- **CORS_ALLOW_ORIGINS**: 특정 도메인만 허용하도록 CORS를 설정했다면, 이를 환경변수나 config에 명시할 수 있습니다. 예를 들어 `CORS_ALLOW_ORIGINS="https://admin.qualijournal.com"` 식으로 설정하여 운영 도메인만 허용하고, 개발 단계에서는 `http://localhost:8000` 등을 추가하는 방식입니다.

- **ENV (환경구분)**: 필요 시 `ENV=prod` 또는 `ENV=dev` 등의 변수를 두어, 개발모드에서는 일부 보안절차를 완화하거나, 디버깅 로그를 늘리는 등의 용도로 사용할 수 있습니다. (예: 개발모드에선 CORS *를 허용, 토큰 체크 비활성 옵션 등을 둘 수 있음.)

- **.env 파일 관리**: 로컬 개발 시에는 프로젝트 루트에 `.env` 파일을 두고 위 변수들을 정의합니다. 이 파일은 `.gitignore`에 포함시켜 **절대 Git에 올라가지 않게** 합니다. 운영 환경에서는 이 .env 내용을 수동으로 Cloud Run 변수에 입력하거나, CI 파이프라인에서 Secrets를 통해 주입합니다 [60]. 예컨대 GitHub

Actions에서는 repository secrets에 `ADMIN_TOKEN`, `DB_URL` 등을 저장해 두고, 배포 시 Workflow에서 Cloud Run에 전달하는 방식을 사용했습니다 [64].

- **시크릿 보관 및 로테이션**: 환경변수로 민감값을 다루면 코드에는 노출되지 않는 장점이 있습니다 [60]. 다만 운영 중 장기적으로 볼 때 키 교체(로테이션)가 필요할 수 있습니다. Admin Token이나 API Key를 교체할 땐, 새 값을 Cloud Run 콘솔에 업데이트하고 서비스 리비전을 새로 배포하면 됩니다. Secrets Manager를 이용하면 보다 체계적인 키 관리가 가능하며, Cloud Run이 시크릿을 주기적으로 최신 값으로 동기화할 수도 있습니다 (현재는 수동 업데이트 방식으로 충분).

- **로그 및 모니터링 보안**: 중요한 것은 **민감정보가 로그에 남지 않도록** 하는 것입니다. 예를 들어, Admin Token 값이나 DB 비밀번호 등이 서버 로그나 에러메시지에 출력되지 않게 주의합니다 [48]. FastAPI 실행 시 `print(config)` 같은 실수를 하면 토큰이 찍힐 수 있으니 조심해야 합니다. Cloud Run 로그를 주기적으로 점검하여 민감한 내용이 없는지 확인하고, 만약 노출되었다면 해당 값을 즉시 변경하는 것이 좋습니다.

- **인증 및 권한 확장**: 현재 Admin 시스템은 단일 토큰으로 보호되고 있어 **일반 사용자 UI와 구분**되어 있습니다 [35]. 추후 운영자가 여러 명으로 늘어나거나 역할별 권한 구분이 필요할 경우, JWT 인증이나 OAuth 연동을 고려할 수 있습니다. 또한 Admin UI 자체를 별도 프록시로 보호하거나, Organization SSO 등을 붙이는 방안도 생각해볼 수 있습니다. 그러나 현 단계에서는 토큰 + CORS 정도로 충분히 간략히 구성되어 있습니다 [35].

요약하면, ADMIN_TOKEN, DB_URL, API_KEY 등이 주요 환경변수이며, 이들은 .env를 통해 주입되고 Cloud Run 배포 시 안전하게 관리됩니다 [56]. 운영 시에는 반드시 실제 컨테이너에 이 값들이 들어갔는지 확인하고 (값이 누락되면 관련 기능이 실패합니다), 추후 DB 연결이나 외부 API 사용 시에도 환경변수를 추가로 정의하여 확장해나가면 됩니다.

6. 데이터 저장 구조 및 DB 연동 가이드

6.1 JSON 기반 데이터 저장 구조

QualiJournal 시스템은 현재 **로컬 JSON 파일**을 데이터 저장소로 사용합니다 [61]. 이는 초기 구현을 간단히 하고, 하루치 데이터가 비교적 적은 규모이기 때문에 가능한 선택입니다. 주요 JSON 파일들과 그 내용은 다음과 같습니다:

- `selected_keyword_articles.json`: 하루 키워드에 대해 **수집된 모든 기사**의 정보를 담은 JSON 파일입니다. 내부에는 기사 목록이 배열로 저장되며, 각 기사 객체는 대략 다음 필드를 가집니다:

```
{
  "id": 5,
  "source": "reddit",
  "title": "Some News Title",
  "url": "http://...article-link",
  "summary": "Original summary text or content snippet",
  "date": "2025-10-11T08:30:00Z",
  "score": 72,
  "approved": false,
  "editor_note": ""
}
```

이 파일은 **자동 수집 단계**에서 생성/갱신되며, 편집자가 GUI를 통해 각 기사에 대해 `approved`를 `true`로 바꾸고 `editor_note`에 코멘트를 남기는 식으로 편집합니다 [1]. JSON 포맷이므로 메모장이나 VSCode로 열어 수동 편집할 수도 있지만, Admin UI를 통해 하는 편이 안전합니다.

- `selected_articles.json`: 최종 **발행 대상 기사 목록**입니다. `selected_keyword_articles.json`에서 `approved=true`인 기사들만 추려서 구조화한 JSON으로, 발행 직전 단계 또는 발행 스크립트(`sync_selected_for_publish.py`)에 의해 생성됩니다 [10]. 이 파일은 Markdown/HTML 뉴스 페이지 생성의 입력으로 사용되며 [3], 발행 완료 후 archive에도 보존됩니다 (예: `archive/2025-10-11_AI.json`). 구조는 필요한 필드만 남기고 정돈되어 있을 수 있습니다:

```
[
 {
  "title": "Some News Title",
  "url": "...",
  "summary_en": "English summary...",
  "summary_ko": "번역된 요약...",
  "editor_note": "편집자 코멘트"
 },
 ...
]
```

(실제 필드는 시스템 구현에 따라 다르나, 요약 및 코멘트 등이 포함됨)

- `config.json`: 시스템 동작에 관한 각종 설정값을 모아둔 JSON 파일입니다. 예를 들어 `features` 섹션에 `require_editor_approval` 또는 `gate_required` (발행을 위한 최소 승인 수) 설정이 있고 [65], `community_filters` 섹션에 커뮤니티 소스에 대한 최소 추천수나 차단 키워드 등의 임계값이 포함될 수 있습니다 [66]. 또한 `external_rss` 섹션에 `gate_required`(외부 공식 기사 풀에서 보충할 기준) 같은 값도 있고, `glossary`나 기타 AI 설정, 섹션 구성 정보 등이 들어있을 수 있습니다. 서버는 시작 시 이 파일을 읽어 전역 설정을 적용하며, `/api/config/...` API로 일부 값을 수정하면 이 파일을 갱신합니다 [15].

- **소스 리스트 파일들**: `official_sources.json`, `community_sources.json`, `paper_sources.json` 등 각 출처 종류별 RSS 피드나 API 목록이 들어있는 파일들입니다. 예를 들어 `official_sources.json`에는 신뢰할 만한 공식 뉴스 사이트들의 RSS URL과 카테고리 정보가 담겨 있고, `community_sources.json`에는 Reddit 등 커뮤니티 소스와 필터 조건(키워드 필터, 최소 추천수 등)이 정의되어 있습니다 [67]. 이러한 파일은 새로운 출처를 추가하거나 변경할 때 편집하여 반영해야 합니다. 운영 중에는 정기적으로 404 오류나는 RSS가 없는지 확인하고 업데이트하는 것이 좋습니다 [68].

- **archive 폴더**: 앞서 설명했듯 과거 발행된 결과물을 날짜별/키워드별로 저장합니다. 여기에는 **Markdown 기사 페이지**, **HTML 기사 페이지**, **해당 날의 JSON 데이터**가 들어갑니다 [4]. 예를 들어 `archive/2025-10-11_AI.md` & `.html` & `.json` 세트가 하나의 패키지입니다. (JSON은 `selected_articles.json`의 아카이브로 볼 수 있습니다.) Archive 폴더 구조는 날짜를 prefix로 하기 때문에 히스토리 기능을 제공하는 데 활용될 수 있습니다. 다만 폴더 구조가 깊지는 않아 수백일치 쌓이면 관리가 힘들 수 있으므로, 월별 서브폴더로 나누거나 DB로 이전하는 것이 장기적으로 필요합니다.

- **기타**: `editor_rules.json` 등 정책성 데이터파일이 있을 수 있습니다. 예컨대 도메인별 기본 점수나 키워드 가중치 등을 정의하는 JSON입니다 [69] [70]. 이 역시 수동/사전 설정파일로서, 점수를 계산하거나 필터링할 때 사용됩니다.

JSON 기반의 워크플로우 장점은 **직관적**이고 **파일 한두 개로 간단히 관리**할 수 있다는 것이지만, 단점은 **동시작업 어려움**, **데이터조회 한계**, **영구보존의 불편함** 등이 있습니다. 그래서 개선 작업에서는 데이터베이스를 도입하는 방향을 병행 검토했습니다 [61].

6.2 PostgreSQL 연동 준비

- **개요**: 현재는 JSON 파일을 사용하고 있으나, 향후 **PostgreSQL** 같은 관계형 데이터베이스로 이관할 준비가 되어 있습니다 [61]. 데이터베이스를 사용하면 다량의 기사 이력도 효율적으로 조회할 수 있고, 여러 키워드를 장기간 아카이브하거나 통계 질의에 유리합니다.

- **스키마 설계**: 백엔드 코드를 키워드 중심으로 개편하면서, 이에 맞는 DB 스키마도 구상 중입니다 [71]. 가령 아래와 같은 테이블 구조를 생각해볼 수 있습니다:

- `keywords` 테이블: 날짜, 키워드, 상태(발행됨/대기) 등을 저장. (하루 하나의 키워드 뉴스 묶음을 식별)
- `articles` 테이블: 기사 개별 정보를 저장. 컬럼에는 id, keyword_id (외래키), source, title, url, summary_en, summary_ko, score, approved, editor_note, timestamp 등.
- `sources` 테이블: 소스별 메타정보 (name, type, url 등) – 기존 JSON 파일 대체.
- `users` 테이블: (추가 가능) 만약 편집자별 관리가 필요하면 사용자 계정 및 권한 정보를 저장.

이러한 구조를 통해 **다중 키워드** 발행이나 히스토리 관리가 수월해질 것입니다. 아직 정확한 스키마는 확정되지 않았으나, JSON에 있던 정보들이 1정규형태로 나눠 들어간다고 볼 수 있습니다.

- **ORM 및 마이그레이션**: 데이터베이스 연동을 위해 SQLAlchemy 같은 ORM을 도입할 수 있습니다. 예를 들어 Pydantic BaseModel과 호환되는 ORM 모델을 정의하고, **Alembic**을 통해 마이그레이션 스크립트를 관리하는 방안을 검토 중입니다 [71]. 엔티티(Entity) 클래스를 만들고, 기존 JSON 로직을 대체/보완하는 방식으로 개발할 계획입니다. 이 과정에서 `DB_URL` 환경변수를 활용하여 DB 연결을 초기화합니다.

- **이행 전략**:

- **병행 운영**: 초기에는 JSON 쓰기와 DB 쓰기를 모두 수행하여 데이터 동기화를 유지하고, 읽기 시에는 DB 우선, 없으면 JSON 백업 사용 등의 전략을 쓸 수 있습니다. 또는
- **일괄 이전**: 특정 시점에 JSON 데이터를 DB로 마이그레이션한 뒤 JSON을 더 이상 사용하지 않도록 전환합니다. 이 경우 과거 archive의 JSON도 DB에 import하여 통합하는 작업이 필요합니다.

현재 문서 시점에서는 아직 기사 선정/발행 데이터는 JSON으로 동작 중이지만, PostgreSQL 전환을 **준비**하고 있는 단계입니다 [72]. 따라서 운영자는 DB가 실제 가동되기 전까지는 JSON 파일의 무결성을 잘 유지해야 합니다. 필요시 `tools/repair_selection_files.py` 등을 사용해 JSON 구조를 교정할 수 있습니다 [10].

- **DB 설정 및 연결 테스트**: 만약 DB를 이미 설정해 두었다면, 서버 `.env`의 `DB_URL`을 통해 **DB 연결 여부를 테스트**해볼 수 있습니다 [73]. (예: 로컬에서 PostgreSQL를 띄워놓고 ENV를 지정한 뒤 서버를 구동하면, 서버 시작 로그에 DB 연결 성공 메시지가 뜨거나, `/api/status` 호출 시 DB에서 읽어온 통계가 나오는 식으로.) 아직 완전한 이행 전이라면, DB를 켰더라도 실질적인 데이터는 없을 수 있으므로 큰 차이는 없겠지만, **DB 연결만이라도 성공하는지** 확인해 두면 향후 전환 시 문제를 줄일 수 있습니다.

- **데이터 영속성과 백업**: JSON에서 DB로 바뀌면, 데이터는 로컬 파일이 아닌 DB에 저장되므로 **백업 전략**도 변경되어야 합니다. JSON 시절에는 archive 폴더를 통째로 압축/백업하는 식으로 관리했다면 [12], DB 사용 후에는 주기적인 DB 덤프나 PITR(Point-In-Time Recovery) 설정 등을 고려해야 합니다. 또한 Cloud Run에서 DB를 쓰려면 Cloud SQL 등 별도 서비스와 연계하거나 VPC 세팅을 해야 하므로 인프라 구성이 추가됩니다.

한편, DB로 옮기더라도 **일부 설정 파일(config, sources 등)은 여전히 JSON/파일로 유지**할 수 있습니다. 예컨대 아주 빈번히 바뀌지 않는 설정은 파일로 두고, 기사/키워드 데이터만 DB로 관리하는 혼합 형태도 가능합니다. 이 부분은 구현상의 편의와 운영패턴에 따라 결정하면 됩니다.

요약: 현 시점에서는 JSON 파일 시스템이 주력이며, DB(PostgreSQL) 연동은 **설계 및 환경 준비가 되어 있는 상태**입니다 [61]. 운영자는 DB_URL 환경변수를 세팅해두고, 추후 코드를 업데이트하여 DB 모드로 전환할 때를 대비하면 됩니다. 전환 시에는 마이그레이션 절차에 따라 기존 JSON 데이터를 DB에 넣고 충분히 테스트한 후 진행하도록 합니다.

7. 통합 테스트 예시 (Pytest 활용)

신규로 추가된 보고서 생성, 요약/번역, 내보내기 등의 기능에 대해 **통합 테스트**를 작성하여 기능이 의도대로 동작하는지 검증합니다 [74]. Pytest 프레임워크를 사용하며, FastAPI의 `TestClient`를 활용해 API 엔드포인트 호출을 모사합니다. 아래는 각 API별 테스트 시나리오와 간단한 코드 예시입니다:

- **사전 준비**: 테스트를 위해 실제 서버 인스턴스를 가져와야 합니다. 예를 들어 `from server_quali import app` 한 뒤 `client = TestClient(app)`를 생성합니다. 또, 일부 테스트에서는 파일 시스템 접근이 있으므로 임시 디렉토리(`tmp_path` fixture)를 활용하거나, 중요한 JSON 파일들의 경로를 테스트용으로 대체(모킹)할 수 있습니다. 또한 ADMIN_TOKEN이 설정된 경우 테스트에서도 헤더를 넣어주어야 하므로, `client = TestClient(app)` 생성 전에 `os.environ['ADMIN_TOKEN'] = "test-token"` 식으로 환경변수를 주거나, testclient 요청 시 헤더를 항상 넣는 등의 처리를 합니다.

- `/api/report` **테스트**: 보고서 생성 호출 시 실제로 Markdown 파일이 만들어지고 응답 JSON에 해당 파일 경로가 포함되는지 확인합니다 [75]. 예:

```
def test_report_generation(tmp_path, monkeypatch):
    # 가상의 selected_articles.json 데이터를 준비
    sample_data = [{ "title": "Test", "url": "http://x", "summary_en": "Sum",
"editor_note": "Note", "approved": True }]
    # monkeypatch를 이용해 selected_articles.json 경로를 tmp_path로 변경하고 샘플 데이터 저
장
    monkeypatch.setattr(server_quali, "SELECTED_ARTICLES_PATH", tmp_path /
"selected_articles.json")
    (tmp_path / "selected_articles.json").write_text(json.dumps(sample_data))

    # API 호출
    response = client.post("/api/report", headers={"Authorization": "Bearer test-token"})
    data = response.json()
    assert response.status_code == 200
    assert data["ok"] is True
    # 파일이 지정된 경로에 생성되었는지 확인
    report_path = data["path"]
    assert os.path.exists(report_path)
    content = Path(report_path).read_text()
    # Markdown 콘텐츠에 sample_data의 내용이 포함되었는지 체크
    assert "Test" in content and "http://x" in content and "Note" in content
```

위 코드는 개념 예시이며, 실제 경로나 모듈 구조에 따라 조정해야 합니다. 핵심은 `/api/report`를 호출한 후:

- 응답 상태가 200인지
- `ok`가 True인지 [75]
- `path` 필드가 존재하며 그 경로에 파일이 생성되었는지

- 파일 내용에 우리가 예상한 내용(기사 제목 등)이 포함되었는지 등을 검증하는 것입니다.

- `/api/enrich/keyword` 및 `/api/enrich/selection` **테스트**: 이들은 실제로 `enrich_cards.py` 실행을 시도하므로, 테스트 환경에 해당 스크립트가 없거나 동작하지 않을 수 있습니다. 대신 백엔드가 **스크립트 실패에도 파일을 생성하는 로직**을 갖고 있으므로 [23], 그 부분을 테스트합니다. 예:

```
def test_enrich_keyword_creates_file(monkeypatch):
    # monkeypatch로 enrich_cards.py 실행 함수를 더미로 대체하여 실행해도 통과되게 함
    monkeypatch.setattr(server_quali, "_run_py", lambda script, args=[]: None)
    # selected_keyword_articles.json에 요약 필드가 있는 샘플 작성
    sample_data = [{ "title": "Hello", "url": "http://y", "summary_en": "Hello world",
"summary_ko": "안녕하세요", "editor_note": "" }]
    monkeypatch.setattr(server_quali, "SELECTED_KEYWORD_PATH", tmp_path /
"ska.json")
    (tmp_path / "ska.json").write_text(json.dumps(sample_data))
    res = client.post("/api/enrich/keyword", headers={"Authorization": "Bearer test-
token"})
    data = res.json()
    assert res.status_code == 200
    assert data["ok"] in (True, False)  # ok가 False일 수도 있지만 일단 200은 온다
    assert "path" in data
    # 경로 파일 존재 및 내용 검증
    md_path = data["path"]
    assert os.path.exists(md_path)
    md_content = Path(md_path).read_text()
    assert "Hello world" in md_content and "안녕하세요" in md_content
```

이 테스트는 요약 스크립트가 실제로 안돌아도(`_run_py`를 noop으로 monkeypatch) **요약 Markdown 파일이 생성**되는지를 확인합니다. `ok` 필드는 스크립트 성공여부에 따라 True/False일 수 있으므로, 우리는 파일 생성과 내용 위주로 검증합니다. `/api/enrich/selection`도 동일한 패턴으로 테스트하면 됩니다.

- `/api/export` **테스트**: Markdown, CSV 내보내기는 브라우저 다운로드를 위한 것이라 테스트가 약간 다릅니다. 응답으로 실제 파일 내용이 오기 때문에, `client.get("/api/export/md")`를 호출하고 `response.text` 또는 `response.content`를 확인합니다. 예:

```
def test_export_csv_format(monkeypatch):
    # 샘플 데이터 준비 (selected_articles.json)
    sample_data = [{ "title": "A", "url": "http://a", "summary_en": "E", "summary_ko":
"K", "editor_note": "N" }]
    monkeypatch.setattr(server_quali, "SELECTED_ARTICLES_PATH", tmp_path /
"sel.json")
    (tmp_path / "sel.json").write_text(json.dumps(sample_data))
    res = client.get("/api/export/csv", headers={"Authorization": "Bearer test-token"})
    assert res.status_code == 200
    csv_text = res.text
    # BOM 확인 (BOM 존재 시 res.text에 \ufeff로 나타날 수 있음)
    assert csv_text.startswith("\ufeff")
```

```
    # CSV 내용에 header나 title 존재 여부 확인
    assert "title" in csv_text.lower() or "A" in csv_text
```

이와 같이 CSV 응답에 BOM이 포함되었는지 (`\ufeff`), 그리고 샘플 기사 "A"의 제목이나 기타 필드가 들어 있는지 검사합니다. Markdown 출력 테스트도 비슷하게, `response.text`에 특정 Markdown 구문(예: `- [ ]` 리스트 표기 등)과 제목이 포함됐는지 확인할 수 있습니다.

- **토큰 인증 테스트**: 보안상 중요한 토큰 인증이 제대로 적용됐는지도 테스트합니다. 예를 들어, 토큰 없이 호출하면 401이 나오는지:

```
def test_auth_token_required():
    res = client.post("/api/report")  # 헤더 없이 호출
    assert res.status_code in (401, 403)
    # 올바른 토큰으로 호출 시 200
    res2 = client.post("/api/report", headers={"Authorization": "Bearer test-token"})
    assert res2.status_code == 200
```

실제 구현에 따라 401 또는 403을 반환하므로 둘 다 허용했습니다 [32].

- **임계값 슬라이더 테스트**: (심화) `/api/config/gate_required`와 `/api/status` 연계를 테스트합니다. 예를 들어 현재 승인된 기사수가 10개, gate_required 기본이 15인 상황에서, API로 gate_required를 10으로 낮추면 `gate_pass`가 True로 바뀌는지를 검증합니다:

```
def test_gate_threshold_change(monkeypatch):
    # monkeypatch 상태: selection_approved=10, initial gate_required=15
    monkeypatch.setattr(server_quali, "get_selection_counts", lambda: (10, 10))  # 총 10
개 중 10개 승인
    server_quali.GATE_REQUIRED = 15
    res1 = client.get("/api/status", headers={"Authorization": "Bearer test-token"}).json()
    assert res1["gate_pass"] is False
    # gate_required를 10으로 변경
    client.patch("/api/config/gate_required", json={"value": 10}, headers={"Authorization":
"Bearer test-token"})
    res2 = client.get("/api/status", headers={"Authorization": "Bearer test-token"}).json()
    assert res2["gate_required"] == 10
    assert res2["gate_pass"] is True
```

위는 개념적인 예시이며, 실제 구현에서는 내부 함수나 전역변수 접근 방식을 테스트에 노출하지 않을 수 있지만, 아이디어는 **슬라이더 조작 -> status 확인** 흐름을 자동화하는 것입니다 [40].

이러한 테스트 케이스들을 **tests/** 폴더에 모아서 관리합니다. 그리고 `pytest`를 실행하여 **전체 테스트가 통과되는지** 확인합니다 [52]. 특히 새로운 기능 관련 테스트(보고서 생성, 요약, 내보내기 등)뿐 아니라 기존 기능(기사 수집, 승인 등 관련된 함수)에 대한 회귀 테스트도 구비하여, 최근 코드 변경이 다른 부분에 영향을 주지 않았음을 보장해야 합니다 [76].

CI 통합

개발 완료 후에는 GitHub Actions CI 파이프라인에 Pytest 단계를 추가하여, **코드 푸시 시 자동으로 테스트를 실행**하고 실패하면 배포를 막는 것이 좋습니다 [51]. 이미 4장 CI/CD에서 언급했듯, 워크플로우 YAML에 Pytest를 수행하는 job을 넣고, `pytest -q` 명령을 실행하도록 합니다. 모든 테스트 케이스가 성공하면 이후 배포 job이 진행되고, 하나라도 실패하면 워크플로우를 중단시켜 **문제있는 코드는 배포되지 않도록** 합니다 [77].

또한 테스트 커버리지 도구를 사용해 주요 로직이 충분히 테스트되고 있는지 살펴보는 것도 권장됩니다 [76]. 예를 들어 coverage.py를 활용해 보고서 생성, 요약 생성 등의 함수가 호출되었는지 확인하고, 필요하다면 추가 테스트를 보완합니다.

통합 테스트 시나리오로 **끝에서 끝(E2E)** 흐름을 검증하는 것도 고려했습니다 [51]. 예컨대, 가상의 기사 20개를 수집 -> 15개 승인 -> `/api/report` 호출 -> `/api/export/csv` 호출까지 일련의 과정을 시뮬레이션하여, 시스템 전체 파이프라인이 문제없이 동작하는지 테스트할 수 있습니다 [51]. 이런 시나리오 테스트는 시간이 걸릴 수 있지만, Admin 시스템 안정성을 높이는 좋은 방법입니다.

8. 운영 체크리스트 및 예외 대응 방법

마지막으로, QualiJournal 관리자 시스템을 안정적으로 운영하기 위해 **주요 체크리스트**와 **예외 상황별 대응 방법**을 정리합니다. 실무자는 아래 항목들을 참고하여 일일 운영 및 장애 대응을 수행하면 됩니다.

8.1 운영 체크리스트 (일상 점검 항목)

- **데일리 발행 전 확인**: 매일 새로운 키워드 뉴스 발행을 준비할 때, **승인된 기사 수**를 확인합니다. 최소 요구 개수(게이트 임계값)를 충족했는지 `/api/status`의 `selection_approved`와 `gate_required`를 통해 확인합니다 [39]. 기본 임계값은 15개이며 [78], 부족할 경우 추가 승인하거나 게이트 임계값을 조정해야 합니다.
- **품질 게이트 통과 여부**: `gate_pass` 값이 False이면 발행을 진행하지 않는 것이 원칙입니다 [39]. 부득이 당일 기사가 부족한 경우 **응급방안**으로 임계값을 일시적으로 낮추거나(require_editor_approval 옵션을 끄거나) [65], 공식 기사 풀 보충 스크립트(`augment_from_official_pool.py`)를 실행하여 기사 수를 늘린 후 진행할 수 있습니다.
- **소스 수집 상태**: 매일 자동 수집이 제대로 이루어졌는지, `selected_keyword_articles.json` 최신 시간을 확인하고, `/api/status`의 `keyword_total` (수집 기사 수) 값이 평소 수준인지 살핍니다. 특정 소스에서 0개 수집되거나 한다면 source 설정에 문제가 없는지 점검합니다.
- **UI 대시보드 KPI 갱신**: 관리자 UI에 표시되는 KPI 지표 (예: 누적 수집, 승인, 커뮤니티/공식 기사 비율 등)가 실제와 일치하는지 확인합니다. UI가 `/api/status`를 주기적으로 부르는지, 또는 수동으로 새로고침해야 하는지에 따라 다르지만, 최소한 발행 직전에는 최신 상태로 갱신해주는 것이 좋습니다.
- **MD/CSV Export 주기적 백업**: 편집자는 필요시 "/api/export/md"나 "/api/export/csv"를 눌러 **현재까지의 기사 목록을 파일로 저장**해 둘 수 있습니다 [27]. 이 기능을 이용하여 중요한 데이터를 주기적으로 백업합니다. 특히 CSV 내보내기는 Excel에서 열어 통계나 분석 자료로 쓰기 좋습니다. (BOM이 포함되어 한글이 깨지지 않음을 다시 한번 확인).
- **Archive 용량 관리**: `archive/` 폴더에 발행물이 쌓이므로, 서버 용량을 주기적으로 체크합니다. 오래된 파일은 별도로 압축 저장하거나, 로컬에 다운로드 후 Cloud Storage 등에 업로드해 두고 본 서버에서는 정리하는 방안을 고려합니다 [12]. Cloud Run 자체는 스테이트리스이지만, 혹시나 NAS 연결 등을 통해 archive를 유지한다면 용량 이슈를 신경써야 합니다.
- **Cloud Run 모니터링**: Cloud Run 콘솔의 **모니터링** 탭에서 CPU 사용률, 메모리 사용량, 요청 지연 시간 등을 관찰합니다. 만약 메모리 부족 경고나 CPU 스로틀링이 보이면, Cloud Run 설정에서 리소스 상향을 검토합니다. 또한 **동시 요청 수**(Concurrency)와 **인스턴스 Scaling** 동향도 파악하여, 너무 자주 스케일 아웃/인 하지 않도록 최소/최대 인스턴스 값을 조정할 수 있습니다.

- **로그 모니터링**: 로그에 지속적으로 오류(traceback)가 발생하지는 않는지 확인합니다. 특히 cron 식으로 돌고 있는 외부 스크립트(run_quali_today.ps1)가 있다면 Cloud Run 로그에는 안 남으므로, 별도로 VM이나 Cloud Scheduler 로그를 봐야 합니다. 하지만 Admin API 내에서의 오류 (예: 토큰 인증 실패 시도, 파일 경로 못찾음 등)는 Cloud Run 로그로 남으니 매일 한 번씩 훑어보는 것이 좋습니다.
- **환경변수/설정 확인**: 운영 중 설정을 바꿨다면 (예: 게이트 임계값 slider로 변경), 다음날 서버 재시작 후에도 적용되었는지 확인합니다. `config.json`에 영구 반영되므로 보통 문제없지만, 간혹 여러 인스턴스 중 하나만 변경된 경우도 있을 수 있으니, 이를 방지하려면 Config 변경은 가급적 한 번에 하고, 변경 후 모든 인스턴스를 재시작하는 것도 한 방법입니다.

8.2 예외 상황 및 대응 방법

- **토큰 인증 오류**: Admin API 호출 시 401/403 에러가 발생할 경우, 사용 중인 토큰이 맞는지 확인합니다. 혹시 최근에 ADMIN_TOKEN을 변경했는데 프론트엔드에 반영하지 않았다면, 새로운 토큰으로 헤더를 업데이트해야 합니다. Unauthorized 에러가 연달아 발생하면 Cloud Run 로그에 해당 기록이 남으므로, **외부에서 토큰 없이 접근 시도가 있지는 않은지** 보안 모니터링도 병행합니다. 필요하면 토큰을 교체하고 알려지지 않도록 관리합니다.

- **API 실패 (보고서/요약 생성 오류)**: `/api/report`나 `/api/enrich/...` 호출이 `ok: false`를 반환하거나 UI에 "실패"로 표시되면, 응답 JSON의 `error` 필드를 확인합니다. 예를 들어 "NameError: 'XYZ' is not defined in make_daily_report.py"와 같은 에러라면 스크립트 문제이므로 코드를 수정해야 합니다. 요약의 경우 "Translation API quota exceeded" 등의 메시지가 올 수 있으니, **API 키 상태**(정상/쿼터초과)를 점검합니다. 그래도 Markdown 결과 파일은 생성되었을 수 있으므로 (idempotent 처리) [23], `archive/reports/` 또는 `archive/enriched/` 폴더를 직접 확인하여 부분적으로 생성된 파일을 수동 보완할 수도 있습니다.

- **데이터 불일치/무결성 문제**: 간혹 편집 도중 JSON이 깨지거나(`selected_articles.json`에서 특정 필드 누락 등)하여 스크립트가 오류를 낼 수 있습니다. 이때는 `tools/repair_selection_files.py`를 실행하여 JSON 포맷을 고치고, `sync_selected_for_publish.py`로 `selected_articles.json`을 재생성합니다 [10]. 이러한 도구들은 응급상황에서 수동으로 CLI로 돌려야 하므로, 운영자는 해당 스크립트들의 사용법 (예: `python tools/repair_selection_files.py`)을 숙지해 두면 좋습니다.

- **발행 강행 필요 상황**: 품질 게이트를 만족하지 못했지만 발행을 해야 하는 상황이라면, `/api/config/gate_required`로 임계값을 현재 승인된 기사 수 이하로 낮춥니다 (예: 15 -> 10으로) [15]. 그러면 `/api/status`에서 `gate_pass:true`로 바뀌므로 발행(스크립트 또는 UI 기능)이 진행될 것입니다. **단, 사후에 이 값을 원래대로 돌려놓는 것**을 잊지 말아야 합니다 [65]. Cloud Run에서는 config.json이 이미 바뀌었으므로, 다시 올려놓으려면 슬라이더를 원위치하거나 수동 PATCH 호출로 설정하면 됩니다.

- **Cloud Run 장애/배포 이슈**: 새 버전 배포 후 API 응답이 비정상이라면, 우선 Cloud Run **로그에서 오류 원인**을 찾습니다. 예컨대 "Module not found" 오류가 있다면 Docker 이미지에 해당 모듈이 빠진 것이므로 `requirements.txt`를 수정하고 재배포해야 합니다. 만약 급한 경우 이전 리비전으로 **트래픽을 롤백**할 수 있습니다 (Cloud Run UI에서 이전 리비전을 100%로 또는 `gcloud run services rollback`). 배포 파이프라인 자체에 문제가 있으면, GitHub Actions 로그를 보고 인증이나 권한 오류를 해결해야 합니다.

- **Database 관련 문제**: 아직 DB를 본격 사용하지는 않지만, 만약 DB_URL을 설정해두고 서버에서 DB 초기화 코드를 실행했다면 **DB 연결 오류**가 발생할 수 있습니다. 이 경우 `.env`의 DB_URL이 올바른지, 데이터베이스 인스턴스가 접근 가능한지(VPC, 권한 등) 확인합니다 [73]. DB를 사용하도록 전환한 후에는 쿼리 실패나 데이터 migration 충돌 등의 이슈가 생길 수 있는데, 이는 해당 시점의 migration 스크립트(log)와 DB 로그를 보며 해결해야 합니다. 기본적으로, **JSON->DB 이행 과정은 서비스 공지 후 짧은 다운타임을 가지고 수행**하는 편이 안전합니다.

- **외부 API 장애**: 뉴스 수집 RSS 피드들이 변경되거나 장애가 생기는 경우, 특정 출처에서 기사가 전혀 안 들어올 수 있습니다. 몇 일치 모니터링하여 지속적으로 0이라면 해당 `official_sources.json` 또는 `community_sources.json`의 URL을 업데이트해야 합니다 [68]. 번역 API 등의 외부 API 장애시에는 해당 기능을 비활성(예: API_KEY 제거)하거나 재시도 로직을 둡니다.

- **편집자 UI 관련**: Admin UI에서 버튼이 동작하지 않거나 JS 오류가 나는 경우, 브라우저 개발자 콘솔(F12)에서 에러 메시지를 확인합니다. 흔한 케이스로 **토큰 미설정으로 인한 401**이나 CORS 블록 등이 있을 수 있습니다. 토큰 문제는 앞서 언급한대로 대응하고, CORS 문제의 경우 Cloud Run에 허용 origin을 추가하거나 (임시로 * 허용), 로컬 개발 중이면 `http://127.0.0.1`과 `http://localhost` 모두 허용하도록 설정을 조정합니다.

- **기능 추가/수정 후 테스트**: 새로운 코드를 배포했을 때는 반드시 **통합 테스트를 다시 한번 실행**해보거나, 최소한 수동으로 모든 주요 기능(보고서 생성, 요약, 내보내기, 슬라이더 조정 등)을 한 바퀴 돌려봅니다. 사람이 UI를 통해 시나리오대로 조작하여 예상대로 동작하면 OK, 중간에 오류가 생기면 즉시 롤백하거나 hot-fix를 배포합니다. 이러한 사전 점검은 운영 사고를 줄이는 중요한 습관입니다.

- **장기 개선 계획 관리**: 운영하면서 수집된 개선 아이디어나 TODO 항목은 별도의 이슈 트래커나 노션 등에 기록해 둡니다. 예를 들어 "AI 요약 고도화", "키워드 추천 알고리즘 도입", "다중 사용자 권한 관리" 등의 장기과제는 당장 구현은 아니더라도 로드맵으로 관리하여 현 시스템 위에서 확장해나갈 수 있도록 합니다 [79]. 현재 Admin 시스템이 안정적으로 구축되었으므로, 이러한 추가 기능들은 차차 검토하며 진행하면 됩니다.

以上 (이상)으로 **QualiJournal 관리자 시스템 기능 개발 및 운영 마무리 가이드북**을 모두 살펴보았습니다. 본 문서는 백엔드 개발자와 운영자가 시스템 구조를 이해하고, 각 기능별 사용 방법과 유지보수 방법을 숙지하도록 작성되었습니다. 실제 운영 중에는 이 가이드북을 참고하여 **순차적으로 체크포인트를 따라가며** 작업하면 큰 무리 없이 시스템을 관리할 수 있을 것입니다. 향후 기능 추가나 변경 시에도 본 문서를 업데이트하여 최신 상태를 반영하면 좋겠습니다. 성공적인 운영을 기원합니다! [80] [81]

---

[1] [2] [3] [4] [7] [8] [9] [10] [11] [12] [65] [66] [67] [68] [78] 1004_3A작업계획report.pdf
file://file-S3G4TQFPei3GeAJcXjN6B3

[5] [6] [13] [14] [15] [16] [18] [20] [22] [23] [24] [25] [26] [36] [37] [38] [39] [41] [43] [62] [63] 작업 완료.txt
file://file_00000000132c6208b90c5e598a587980

[17] [21] [28] [30] [42] 1010_1_나머지 계획_퀄리저널 관리자 시스템 개선 작업 인수인계 보고서.pdf
file://file-XBYbxoGjLn4pSrN6QHLGAd

[19] [27] [29] [31] [32] [33] [34] [35] [40] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [64] [71] [72] [73] [74] [75] [76] [77] [79] [80] [81] 1011_1퀄리저널 관리자 시스템 (Admin API) 개선 작업 인수인계 보고서.pdf
file://file-MeqUbXwLc5KBspHeGiMAGN

[69] [70] 1003_5A퀄리뉴스를 전면적으로 리팩터링하여 안정적이고 확장 가능한 시스템으로 구축하려면.pdf
file://file-C1X8pMSvAVd6Xs392FEnwq