ChatGPT

QualiJournal 관리자 시스템 가이드

소개 (Overview)

퀄리저널(QualiJournal) 시스템은 **"하루 하나의 키워드"**를 중심으로 뉴스, 블로그, 학술자료, 커뮤니티 글 등 다양한 출처의 콘텐츠를 자동 수집·큐레이션하고, 편집자의 검수를 거쳐 매일 키워드 특집 뉴스를 발행하는 플랫폼입니다 [1]. 이 가이드는 **개발자**와 **운영자(편집자)** 모두를 대상으로 하며, 시스템의 배포부터 운영, 유지보수까지 **처음부터 끝까지** 독립적으로 수행할 수 있도록 상세한 지침을 제공합니다.

본 문서에서는 Cloud Run을 이용한 배포 절차, 환경 변수와 비밀 키 관리, 주요 API 사용법(cURL 및 fetch 예시 포함), 관리자 UI 활용 방법과 개선 사항, 자동화 테스트, CI/CD 구성, 보안 및 로깅, 백업 연계, 향후 데이터베이스 전환 계획, 그리고 **일일 운영 루틴(예: 아침 시스템 점검, 보고서 생성 절차)**을 다룹니다. 짧은 단락과 목록으로 핵심 내용을 정리하여 개발자와 운영자 모두가 쉽게 활용할 수 있도록 작성되었습니다.

Cloud Run 배포 절차

Google Cloud Run을 통해 QualiJournal 시스템을 **컨테이너 기반** 서버리스 애플리케이션으로 배포할 수 있습니다. Cloud Run은 컨테이너 이미지를 실행하며 자동 스케일링과 관리형 인프라를 제공하므로, 우리 시스템의 배포 및 운영에 용이합니다 [2]. 배포 방법은 **소스 코드 직접 배포**와 **이미지 배포** 두 가지로 나뉘며, 환경 변수(.env) 설정도 포함되어야 합니다. 아래에 실무에서 활용 가능한 CLI 중심 절차를 설명합니다.

1. 환경 준비 및 프로젝트 설정

- **gcloud CLI 인증:** 터미널에서 `gcloud auth login`으로 GCP에 로그인하고, `gcloud config set project <PROJECT_ID>`로 적절한 프로젝트를 선택합니다.
- **Cloud Run API 활성화:** GCP 콘솔에서 Cloud Run API를 사용 설정하거나, CLI로 `gcloud services enable run.googleapis.com`를 실행합니다.
- **Dockerfile 확인:** QualiJournal 애플리케이션의 프로젝트 디렉터리에 `Dockerfile`이 존재해야 합니다. Dockerfile은 애플리케이션 실행 환경을 정의하며, Cloud Run 배포 시 사용됩니다. (예: Python 기반 FastAPI 서버와 정적 파일을 서빙하는 설정 포함)

2. 소스 코드 기반 배포 (Build & Deploy from Source)

별도의 이미지를 수동으로 만들지 않고, **Cloud Build**를 통해 소스에서 직접 Cloud Run에 배포할 수 있습니다. 1. 터미널에서 프로젝트 디렉터리로 이동한 후, 아래 명령을 실행합니다:

```
gcloud run deploy qualijournal-service \
  --source . \
  --region=asia-northeast3 \
  --allow-unauthenticated \
  --update-env-vars "ADMIN_TOKEN=<발행한토큰>,API_KEY=<API키>,DB_URL=<DB연결문자열>,GCS_BUCKET=<백업버킷명>"
```

- `qualijournal-service`는 Cloud Run 서비스 이름 예시입니다. 원하는 이름으로 지정 가능합니다. - `--source .` 옵션은 현재 디렉터리를 빌드하여 배포합니다. (Cloud Build가 Dockerfile을 이용해 이미지를 생성함) -

`--region` 은 가까운 리전을 지정합니다 (예: 서울은 `asia-northeast3`). - `--allow-unauthenticated` 는 **테스트나 내부망 용도**가 아닌 **웹에서 바로 접근해야 하는 관리자 UI**라면 설정합니다. (API 보호를 위해 ADMIN_TOKEN을 사용하므로 공개 접근을 허용해도 토큰이 없는 요청은 처리되지 않습니다.) - `--update-env-vars` 를 통해 필요한 환경 변수를 설정합니다. 각 키=값 쌍은 쉼표로 구분합니다. (ADMIN_TOKEN, API_KEY 등 자세한 설명은 아래 `.env 구성` 섹션 참고) 2. 명령 실행 후 Cloud Build가 소스를 빌드하고, Cloud Run에 배포합니다. 완료되면 Cloud Run 서비스 URL이 출력됩니다. 해당 URL이 **관리자 UI 접속 주소**가 됩니다 (예: `https://qualijournal-service-xxxxxx.run.app`).

3. 컨테이너 이미지 기반 배포 (Pre-built Image)

이미지 배포는 로컬이나 CI 도구에서 도커 이미지를 빌드하여 **Artifact Registry** 혹은 **Docker Hub** 등에 업로드한 후 Cloud Run에서 실행하는 방식입니다. 1. **Docker 이미지 빌드:** 로컬 환경에서 다음 명령으로 이미지를 빌드합니다:

```
docker build -t asia-northeast3-docker.pkg.dev/<PROJECT_ID>/containers/
qualijournal:latest .
```

위 명령은 `<PROJECT_ID>` GCP 프로젝트의 Artifact Registry에 `containers` 라는 리포지토리를 만들었다고 가정하고, `qualijournal:latest` 태그로 이미지를 빌드합니다. (리포지토리가 없다면 미리 Artifact Registry에 도커 리포지토리를 생성하세요.) 2. **이미지 푸시:** 빌드한 이미지를 Artifact Registry에 푸시합니다:

```
docker push asia-northeast3-docker.pkg.dev/<PROJECT_ID>/containers/
qualijournal:latest
```

이 과정에서 Artifact Registry에 대한 인증 세팅이 필요할 수 있습니다. (`gcloud auth configure-docker` 명령으로 gcloud 자격증명을 Docker에 등록) 3. **Cloud Run 서비스 생성/업데이트:** 푸시된 이미지를 사용하여 Cloud Run에 배포합니다:

```
gcloud run deploy qualijournal-service \
  --image asia-northeast3-docker.pkg.dev/<PROJECT_ID>/containers/
qualijournal:latest \
  --region=asia-northeast3 \
  --allow-unauthenticated \
  --update-env-vars "ADMIN_TOKEN=<발행한토큰>,API_KEY=<API키>,DB_URL=<DB연결문자열
>,GCS_BUCKET=<백업버킷명>"
```

- `--image` 플래그에 앞서 푸시한 컨테이너 이미지 경로를 지정합니다. - 나머지 옵션은 소스 배포와 동일합니다. 4. **배포 확인:** 배포 후 출력된 URL로 접속하여 **관리자 UI** 화면이 뜨는지 확인합니다. 최초 접속시 토큰 입력 등이 요구될 수 있습니다 (자세한 것은 UI 섹션 참조). 또한 `gcloud run services describe qualijournal-service --region=asia-northeast3` 명령으로 설정된 환경 변수나 트래픽 구성을 확인할 수 있습니다.

4. 환경 변수 설정 및 .env 반영

Cloud Run 배포 시 지정한 환경 변수들은 Cloud Run 컨테이너에 주입됩니다. 이러한 **설정값과 비밀 키들은 .env 파일**에도 정리하여 로컬 개발 및 CI에서 활용합니다. (.env 구성 상세는 다음 섹션에서 다룹니다.) 배포 후 Cloud Run 콘솔의 **Variables & Secrets** 탭에서 값들이 올바르게 설정되었는지 검증하세요.

- **Secret Manager 연동:** 민감한 값은 Cloud Run에 직접 입력하기보다 GCP Secret Manager에 저장하고 참조하는 것을 권장합니다. Cloud Run 배포 명령에서 `--set-secrets="API_KEY=projects/<PRJ>/secrets/<NAME>:latest"` 와 같이 시크릿을 연결하면, 런타임에 해당 환경변수가 시크릿 값으로 세팅됩니다.
- **배포 이미지 롤백:** 새로운 이미지 배포에 문제가 있을 경우, Cloud Run 콘솔에서 이전 리비전을 선택해 롤백할 수 있습니다. 또한 여러 리비전에 트래픽을 분배해 **카나리 배포**를 할 수도 있습니다.

5. 배포 파이프라인 (선택 사항)

Cloud Run은 CI/CD와 연계하여 자동 배포될 수 있습니다. (자세한 CI/CD 구성은 후술) 개발자가 코드를 main 브랜치에 머지하면, **CI 파이프라인이 도커 이미지를 빌드 및 테스트 후** 자동으로 Cloud Run에 배포하도록 설정할 수 있습니다 [2]. 이를 통해 배포 일관성을 높이고 수동 개입을 최소화할 수 있습니다.

⚙ **팁:** 배포가 성공하면 Cloud Run 서비스 URL이 제공되며, 이 URL을 편집자가 **관리자 웹 UI**로 사용합니다. 필요에 따라 사용자 도메인을 연결하고 SSL 인증서를 설정할 수 있습니다 (예: Cloud Run 커스텀 도메인 매핑). 운영 환경과 개발(staging) 환경을 분리하려면 각기 다른 서비스나 프로젝트를 사용하고, .env/환경변수 구성도 각각 관리하면 됩니다.

환경 변수(.env) 구성 및 비밀 키 관리

QualiJournal 시스템은 여러 **환경 변수**를 활용하며, 이는 `.env` 파일로 관리하거나 Cloud Run 등의 배포 설정에 주입됩니다. 올바른 환경 변수 구성은 보안과 기능 양쪽에서 중요합니다. 아래는 주요 환경 변수와 관리 방법입니다.

1. 주요 환경 변수

- `ADMIN_TOKEN`: 관리자용 API 인증 토큰입니다. **관리자 UI 및 API 호출을 보호**하기 위해 사용됩니다. 서버는 클라이언트로부터 전달받은 토큰이 이 값과 일치하는지 검증하여 API 요청을 허용합니다. 추측이 어려운 임의의 문자열로 설정하고, 정기적으로 변경하는 것이 좋습니다.
- `API_KEY`: 요약/번역 등 **외부 AI 서비스**(예: OpenAI GPT, DeepL API, Google Translate API 등)가 있다면, 그에 대한 API 키를 지정합니다. 뉴스 카드 요약/번역 API가 이 키를 사용해 외부 서비스에 요청합니다. 해당 키 역시 외부에 노출되지 않도록 안전하게 관리해야 합니다.
- `DB_URL`: **데이터베이스 연결 문자열**입니다. 추후 데이터베이스를 사용하거나 현재 일부 기능에서 DB를 사용한다면, DB 접속 정보를 환경 변수로 지정합니다. 예를 들어 PostgreSQL의 경우 `postgresql://user:password@host:5432/database` 형태가 될 수 있습니다. (현재 시스템은 기사 데이터를 JSON 파일에 저장하지만 [3], 향후 DB로 전환 시 이 변수가 활용됩니다.)
- `GCS_BUCKET`: **Google Cloud Storage 버킷 이름**으로, 백업 및 아카이브 파일을 저장하는 곳입니다. 발행된 결과물(MD, HTML, JSON)을 이 버킷에 저장하여 영구 보존 및 백업합니다. Cloud Run에서는 해당 버킷에 쓰기 위해 서비스 계정에 권한(IAM에서 Storage 객체 Admin 등)이 부여되어야 합니다.

이 밖에도 이메일 알림 설정이나 기타 구성 옵션이 있다면 추가 환경 변수를 활용할 수 있습니다. 예를 들어 Slack 웹훅 URL, SMTP 서버 정보 등이 있을 수 있습니다 (해당 기능이 있을 경우).

```
# .env 예시 (실제 값은 보안을 위해 노출 금지)
ADMIN_TOKEN=super-secret-admin-token
```

```
API_KEY=abcdef1234567890abcdef    # 예: OpenAI API Key
DB_URL=postgresql://user:pass@34.64.xxx.yyy:5432/qualidb
GCS_BUCKET=qualijournal-archive
FEATURE_REQUIRE_APPROVAL=true    # 예: 편집자 승인 필요여부 (config.json으로 관리 가능)
```

주의: `.env` 파일은 **절대 소스 레포지토리에 커밋하지 말고**, 배포 시에는 CI/CD 또는 Cloud Run 환경 변수 설정을 통해 주입하도록 합니다. 로컬 개발 시에는 `.env` 파일을 로딩하여 (`python-dotenv` 등을 활용) 환경 변수를 세팅하고 앱을 실행합니다.

2. 비밀 키 관리 및 보안 조치

- **Secret Manager 사용:** GCP의 Secret Manager를 활용하여 ADMIN_TOKEN, API_KEY 등을 저장하고, Cloud Run에서 참조하는 것이 안전합니다. 이렇게 하면 평문 비밀값이 Cloud Run 설정에 직접 드러나지 않고, Secret Manager 권한이 있는 서비스만 값에 접근합니다.
- **권한 관리:** 운영자가 ADMIN_TOKEN을 다루는 경우, 해당 토큰을 **안전한 채널**로 전달하고 공유를 최소화하세요. 만약 토큰이 노출되었다면 즉시 새 값으로 변경(.env 갱신 및 Cloud Run 재배포)하고, 기존 토큰으로는 더 이상 요청이 처리되지 않도록 서버 측 값을 교체하세요.
- **회전(토큰 교체):** 주기적으로 ADMIN_TOKEN과 API_KEY를 갱신하는 것을 권장합니다. 새 토큰으로 서버 설정을 업데이트한 후, **관리자 UI에서 토큰 정보를 최신으로 입력**해야 계속 액세스가 가능합니다.
- **환경 분리:** 개발/스테이징 환경과 프로덕션 환경의 키를 분리하고, 각 환경별 별도 .env와 Secret Manager 항목을 사용하세요. 예를 들어 테스트용 API_KEY와 운영 API_KEY를 혼용하지 않도록 구분합니다.
- **로깅 주의:** 중요 키나 민감 정보는 로그에 남기지 않도록 합니다. 예를 들어 요청을 디버깅할 때 토큰 값이나 DB 비밀번호 등이 출력되지 않게 신경써야 합니다.

환경 변수와 비밀 키의 철저한 관리는 시스템 보안의 기본입니다. **배포 전후로 환경 변수가 정확히 반영되었는지** 확인하고, 키 변경 시 운영자에게 공지하여 새로운 키로 UI를 업데이트하도록 합니다.

일일 보고서 자동 생성 API (`/api/report`)

QualiJournal 시스템의 핵심 작업 중 하나는 **매일 특정 키워드에 대한 뉴스 보고서를 자동으로 생성**하는 것입니다. `/api/report` 엔드포인트는 이러한 일련의 수집→요약→발행 준비 과정을 트리거하는 API입니다. 이 API를 통해 **하루치 키워드 뉴스 수집 및 초안 생성 작업**을 자동화할 수 있습니다.

기능 개요

- **수집(Collect):** 지정된 키워드로 관련 콘텐츠를 모든 출처(공식뉴스, 학술, 커뮤니티 등)에서 수집합니다 [4]. 내부적으로 RSS 크롤러, API 호출 등을 수행하여 raw 데이터를 모읍니다.
- **재구성(Rebuild):** 수집된 자료들을 표준 구조로 통합하여 `selected_keyword_articles.json` 작업 파일을 생성합니다 [5]. 이 파일(또는 해당 메모리 객체)에 키워드와 기사 리스트(제목, 요약, 출처, 날짜, 점수 등)가 정리됩니다.
- **보충(Augment):** 수집 결과나 승인된 기사 수가 부족할 경우, **공식 기사 풀**에서 추가 기사를 채워 총 기사 수를 보정합니다 [6] [7]. 이는 하루 최소 발행 기준을 충족하도록 돕는 절차입니다 (예: 상위 점수 기사들을 자동 승인 처리하거나 추가 기사로 15개 이상 확보).
- **요약/번역 (Enrich):** 수집된 기사들 중 주요 문장을 추출하고 한국어 요약 및 번역을 생성합니다 [8] [9]. 영어 등 외국어 기사에는 AI 번역을 활용하여 한국어 **뉴스 카드 요약**을 만듭니다.
- **동기화(Sync) 및 결과 저장:** 승인 플래그와 편집자 코멘트 필드를 포함한 구조로 데이터를 정리하고, **발행본 (`selected_articles.json`)**을 준비합니다 [10]. 자동 모드에서는 일정 기준으로 (예: 점수 상위 15개) 승인 처리될 수 있습니다.

- **출력(아카이브 임시 저장):** 최종 발행될 Markdown/HTML 뉴스 페이지 초안을 생성합니다 [11]. 이 때 실제로 출판을 완료하지 않고, **미리보기용 초안** 상태로 저장하거나 반환합니다. (자동 모드에서는 바로 발행까지 진행할 수도 있습니다. 설정에 따라 다름)

사용 방법

운영자는 `/api/report` 엔드포인트를 수동으로 호출하거나 (개발 테스트 혹은 즉시 실행 용도) **Cloud Scheduler** 등을 통해 **매일 정해진 시간에 자동 호출**하도록 설정할 수 있습니다 [12]. 해당 API는 **POST 메서드**로 호출하며, 키워드를 파라미터로 전달해야 합니다 (하루에 하나의 키워드 기준).

- **cURL 예시:** 매일 "인공지능" 키워드로 보고서를 생성하는 경우

```
curl -X POST "https://qualijournal-service-xxxxxx.run.app/api/report?keyword=인공지능" \
  -H "Authorization: Bearer <ADMIN_TOKEN>"
```

위 예시는 POST 요청으로 키워드 **인공지능**에 대한 수집/생성을 실행합니다. `Authorization` 헤더에 사전에 발급된 ADMIN_TOKEN을 포함해야 하며, 토큰이 맞지 않으면 401 Unauthorized 응답을 받습니다. (Alternatively, 토큰을 `x-admin-token` 등의 헤더로 구현했다면 그에 맞게 사용)

- **fetch 예시 (JavaScript):** 프런트엔드(관리자 UI)에서 키워드를 입력 받고 요청하는 경우

```
const keyword = "인공지능";
fetch(`/api/report?keyword=${encodeURIComponent(keyword)}`, {
 method: 'POST',
 headers: { 'Authorization': `Bearer ${localStorage.getItem('ADMIN_TOKEN')}` }
})
 .then(response => response.json())
 .then(data => {
  console.log("수집된 기사 목록:", data.articles);
  // 기사 리스트를 상태에 저장하여 UI에 표시
 });
```

UI에서는 ADMIN_TOKEN을 localStorage 등에서 불러와 헤더에 첨부하고 `/api/report`를 호출합니다. 응답으로 **수집 및 처리된 기사 리스트** (`data.articles`)가 JSON 형태로 반환됩니다. 각 기사 항목에는 예를 들어 아래와 같은 필드들이 포함될 것입니다:

```
{
 "id": 3,
 "title": "OpenAI, 새로운 GPT-4 모델 발표",
 "source": "공식뉴스",
 "date": "2025-10-14",
 "summary": "OpenAI는 GPT-4의 향상된 버전을 공개했습니다...",
 "lang": "en",
 "score": 87,
 "approved": false,
 "editor_note": ""
}
```

(위는 예시 데이터 구조입니다. `approved`와 `editor_note` 등은 편집자 확인 후 변경됩니다.)

자동 스케줄링 (Cloud Scheduler 연계)

지속적인 자동 발행을 위해 GCP **Cloud Scheduler**를 활용할 수 있습니다. Cloud Scheduler를 설정하여 `/api/report`를 매일 원하는 시간에 호출하면, 해당 시각에 자동으로 수집 및 초안 생성이 이뤄집니다. 예를 들어 오전 6시에 매일 실행: - GCP Cloud Scheduler에서 새 작업을 만들고, 빈 JSON 페이로드로 POST 요청을 Cloud Run 서비스의 `/api/report` URL에 보내도록 설정합니다. - 이 때 인증이 필요하므로, **Cloud Scheduler의 서비스 계정에 Cloud Run Invoke 권한**을 주고, Cloud Run 서비스의 **인증 필요 설정을 유지**하는 방법이 있습니다. 또는 Cloud Scheduler에서 호출 시 헤더에 토큰을 포함하도록 설정할 수도 있습니다. (UI 외부의 자동 호출이므로 보다 안전한 IAM 기반 인증 방법을 고려해볼 수 있습니다.)

예외 처리 및 품질 게이트

- `/api/report` 수행 중 오류 발생 시 (예: 외부 API 실패나 수집 단계 네트워크 에러) 서버는 **적절한 HTTP 오류 코드**와 메시지를 반환합니다. 예를 들어 수집 단계 실패 시 502 Bad Gateway 혹은 500 오류와 원인 로그를 남깁니다. Cloud Run **로그에서 상세 내역**을 확인하고 필요시 재시도할 수 있습니다.
- **품질 게이트(QG):** 발행을 위해 기본적으로 **최소 15개의 기사가 승인**되어야 합니다 [13] [14]. `/api/report`는 이 기준을 만족하지 못할 경우 완전한 발행 단계까지 진행하지 않거나, 자동 모드에서는 최고 점수 기사들을 자동 승인 처리하여 기준을 충족시킵니다 [15] [16]. 응답에는 승인된 기사 수나 부족 등의 정보가 포함될 수 있습니다. 운영자는 이 기준에 못 미치는 경우 수동으로 기사를 추가하거나, 긴급시 **설정 옵션**인 `require_editor_approval=false`로 변경하여 (토큰이 있는 내부 API 또는 config 설정) 임시로 발행을 강행할 수 있습니다 [17] [18]. 단, 응급 발행 후에는 해당 옵션을 True로 복귀시켜 품질 게이트를 원상태로 돌려놓아야 합니다.
- **응답 데이터 활용:** 이 API가 반환한 기사 목록은 **관리자 UI에서 편집자가 검토 및 승인 작업을 수행할 때 기본 데이터**로 쓰입니다. 만약 자동으로 모든 기사가 승인된 상태로 반환되었다면 운영자는 바로 미리보기/발행 단계로 넘어갈 수 있고, 그렇지 않다면 UI에서 개별 승인 작업이 필요합니다.

요약하면 `/api/report` API는 **일일 키워드 뉴스 생성의 출발점**입니다. 개발자는 이 엔드포인트를 구현함에 있어 파이프라인 각 단계(수집→재구성→보충→요약→출력)를 차례로 수행하도록 하고, 적절한 로그를 남겨 디버깅 가능하게 해야 합니다 [19]. 운영자는 이 API를 자동/수동으로 활용하여 매일의 뉴스 초안을 얻고, 이후의 편집 작업을 진행하게 됩니다.

뉴스 카드 요약/번역 API (`/api/enrich/keyword`, `/api/enrich/selection`)

QualiJournal의 차별화된 기능은 다국어 및 장문의 콘텐츠를 **한국어 뉴스 카드 형태로 요약 및 번역**하여 편집자가 손쉽게 활용할 수 있게 하는 것입니다. 이를 지원하기 위해 두 개의 API 엔드포인트를 제공합니다: - `/api/enrich/keyword` – **키워드 전체 기사들에 대한 일괄 요약/번역** - `/api/enrich/selection` – **선택된 텍스트 또는 특정 기사에 대한 맞춤 요약/번역**

`/api/enrich/keyword` – 키워드 뉴스 일괄 요약/번역

이 엔드포인트는 현재 선택된 키워드의 **모든 기사 카드에 대해 한국어 요약과 필요한 경우 번역을 수행**합니다. 일반적으로 `/api/report`로 초기 수집을 완료한 후, 편집자는 아직 요약되지 않은 원문(특히 영어) 기사들을 보게 됩니다. 이때 `/api/enrich/keyword`를 호출하면: - 각 기사 객체의 주요 문장(핵심 문장 2~3개)을 추출하고 [9], OpenAI나 DeepL 등의 API를 통해 **한국어로 자연스러운 요약문**을 생성합니다 [20]. (이미 한국어 기사라면 내용 간추림 정도만 수행) - 영문 기사의 경우 번역 API를 사용하여 제목이나 중요 키워드를 번역하고, 요약문 역시 한국어로 생성합니다. Glossary(용어집)가 있다면 미리 정의된 전문용어 번역을 적용합니다 [21]. - 생성된 **요약문과 번역된 필드**를 각

기사 데이터에 추가하거나 갱신합니다. 예를 들어 각 기사 데이터에 `summary_kr` 혹은 기존 `summary` 필드 갱신, `translator_note` 등의 필드가 채워질 수 있습니다.

사용 예: (편집자 UI에서 "일괄 요약 생성" 버튼을 눌렀다고 가정)

```
curl -X POST "https://.../api/enrich/keyword?keyword=인공지능" \
  -H "Authorization: Bearer <ADMIN_TOKEN>"
```

별도의 요청 바디 없이, 쿼리 파라미터 또는 URL 경로로 키워드를 지정합니다. 서버는 해당 키워드의 기사 목록을 찾아 일괄 처리하고, 성공 시 갱신된 기사 리스트나 처리 결과를 반환합니다. 응답 예시 (요약된 기사 2건):

```
{
 "keyword": "인공지능",
 "enriched": 2,
 "articles": [
  {
   "id": 3,
   "title": "OpenAI, 새로운 GPT-4 모델 발표",
   "summary": "OpenAI가 GPT-4의 개선된 버전을 발표했습니다. 이 모델은 이전보다 빠르고 정교한 응답을 제공하며, 이미지 입력도 처리할 수 있습니다.",
   "lang": "en",
   "translated": true
  },
  {
   "id": 7,
   "title": "KAIST, AI 연구 센터 개소",
   "summary": "KAIST는 인공지능 연구를 위한 새로운 센터를 열었습니다. 이 센터는 자율주행, 의료 AI 등 다양한 분야의 혁신을 목표로 합니다.",
   "lang": "ko",
   "translated": false
  }
 ]
}
```

위 응답에서 `enriched: 2`는 두 건의 기사가 요약/번역되었음을 의미합니다. 영문 기사(id 3)는 번역되어 summary가 한국어로 채워졌고 `translated: true`로 나타납니다. 한편 원래 한국어 기사(id 7)는 번역은 불필요했지만 핵심 내용이 summary로 정제되었습니다.

fetch 예시: 프런트엔드에서는 호출 후 진행 상황을 표시하기 위해:

```
setLoading(true);
fetch(`/api/enrich/keyword?keyword=${currentKeyword}`, {
 method: 'POST',
 headers: {'Authorization': `Bearer ${token}`}
})
 .then(res => res.json())
 .then(result => {
  setLoading(false);
```

```
  console.log("요약 완료:", result.enriched, "건 처리됨");
  updateArticles(result.articles); // 상태 갱신하여 UI에 요약문 표시
 })
 .catch(err => {
  setLoading(false);
  alert("요약 중 오류 발생: " + err.message);
 });
```

이 때 UI는 로딩 스피너를 표시하여 진행 중임을 알리고, 완료 후에 스피너를 숨기는 식으로 구현합니다 (아래 UI 개선 섹션 참조).

`/api/enrich/selection` – 선택 텍스트/기사 요약/번역

운영자가 개별 기사 또는 특정 텍스트를 선택하여 **부분적인 요약/번역**을 원할 때 사용하는 엔드포인트입니다. 주요 사용 시나리오는: - **기사 본문 번역:** 편집자가 어떤 기사 상세 내용을 읽는 중 특정 문단을 번역하고 싶을 때, 해당 텍스트를 선택하여 번역 요청. - **개별 기사 요약 재생성:** 자동 요약 결과가 미흡한 경우, 개별 기사에 대해 재요약/번역 시도.

이 API는 일반적으로 **요청 바디에 텍스트**를 전달하도록 설계됩니다. 선택한 원문 텍스트를 보내면, 시스템이 해당 텍스트를 한국어로 번역하거나 요약해 줍니다. 만약 특정 기사 ID를 기준으로 동작한다면, 서버가 해당 기사의 원문 전체를 크롤링/조회하여 요약할 수도 있습니다.

- **cURL 예시 (텍스트 직접 전달):**

```
curl -X POST "https://.../api/enrich/selection" \
   -H "Authorization: Bearer <ADMIN_TOKEN>" \
   -H "Content-Type: application/json" \
   -d '{"text": "AI is transforming the world of technology, creating new opportunities and challenges..."}'
```

위 요청은 본문에 영어 문장이 담긴 JSON을 보내어, 그에 대한 한국어 번역/요약을 받고자 합니다. 응답 예시:

```
{
 "translated_text": "AI는 기술의 세계를 변화시키고 있으며, 새로운 기회와 도전을 만들어내고 있습니다...",
 "summary_text": "인공지능(AI)이 기술 분야에 혁신을 일으키며 기회와 도전을 창출하고 있다..."
}
```

여기서 `translated_text`는 전문 번역 결과, `summary_text`는 요약문일 수 있습니다. (시나리오에 따라 번역 또는 요약 중 하나만 수행하게 할 수도 있습니다. 예를 들어 매개변수로 `?mode=translate` 또는 `?mode=summary` 등을 지원 가능.)

- **fetch 예시:** (특정 기사 다시 요약 예)

```
const originalParagraph = "This is a long paragraph in English that needs translation...";
fetch('/api/enrich/selection', {
 method: 'POST',
 headers: {
```

```
    'Authorization': `Bearer ${token}`,
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ text: originalParagraph })
})
  .then(res => res.json())
  .then(data => {
    console.log("번역 결과:", data.translated_text);
    console.log("요약 결과:", data.summary_text);
    // 필요에 따라 UI에 표시하거나 editor_note 등에 반영
  });
```

이와 같은 기능은 편집자가 원문을 깊이 이해하거나 요약문을 다듬는 데 도움을 줍니다. 번역 API 호출이 외부 서비스에 의존한다면, **응답 시간이 길어질 수 있으므로 UI에서 개별 스피너 또는 진행상태 표시**를 해주는 것이 좋습니다.

에러 처리 및 참고 사항

- 외부 API 오류 시: 요약/번역은 OpenAI나 기타 API에 의존하므로, 요금 한도 초과나 네트워크 장애 시 실패할 수 있습니다. 서버는 해당 오류를 캐치하여 502나 503 오류와 메시지를 반환합니다. UI에서는 이를 감지해 사용자에게 "요약 서비스 응답이 지연되고 있습니다" 등의 안내를 표시합니다.
- 요약 내용 편집: 자동 생성된 요약문은 **편집자에 의해 최종 수정될 수 있음**을 염두에 두세요. UI 상에서 summary 필드를 직접 편집 가능하도록 하거나, 편집자 코멘트(`editor_note`)에 기본값으로 요약을 넣고 수정하게 하는 전략도 활용합니다 [22].
- 다중 호출 제어: `/api/enrich/keyword`는 비용이 큰 작업일 수 있으므로, **동시에 두 번 이상 호출되지 않도록** 서버에서 제어하거나 UI 버튼을 디세이블할 필요가 있습니다. 또한 진행 중임을 나타내는 **스피너 UI 표시**로 사용자 혼란을 막습니다.

이처럼 enrich API들은 수집된 원자료를 **한국어로 풍부하게 가공**하여 QualiJournal만의 핵심 가치인 이해하기 쉬운 뉴스 카드로 변환하는 역할을 합니다. 개발자는 API 키 관리 및 외부 API 호출 부분을 견고하게 구현하고, 운영자는 이러한 기능을 적재적소에 활용하여 효율적인 편집 작업을 할 수 있습니다.

Export 기능 API (`/api/export/md`, `/api/export/csv`)

편집과 검토를 마친 후에는 **최종 결과물을 내보내기(export)** 하여 발행하거나 공유해야 합니다. QualiJournal 시스템에서는 Markdown 형식과 CSV 형식으로 결과를 추출하는 두 가지 API 엔드포인트를 제공합니다: - `/api/export/md` – **Markdown 포맷 뉴스 파일** 생성 - `/api/export/csv` – **CSV 포맷 데이터 내보내기**

이를 통해 편집자는 최종 확인 및 백업을 하고, 개발자는 다른 시스템 연동이나 추가 가공을 쉽게 할 수 있습니다.

`/api/export/md` – 최종 Markdown 내보내기

Markdown 형식은 QualiJournal 뉴스 페이지의 원본 형태입니다 [23]. `/api/export/md`를 호출하면 현재 키워드의 승인된 기사들로 구성된 **최종 발행 Markdown 콘텐츠**를 반환합니다. 이 Markdown에는 보통 다음 요소가 포함됩니다: - 키워드 및 날짜 헤더 (뉴스 제목 부분) - 기사 리스트 (섹션별 혹은 점수순 정렬) - 각 기사 카드: 제목, 출처, 날짜, **요약문**(번역된 경우 한국어 요약), 편집자 코멘트 등 - 하단에 출처 링크 모음이나 편집자 노트 섹션 (필요시)

사용 방법: - 편집에서 **최소 15개 기사**를 승인하고, 필요시 모든 요약/번역을 완료한 상태에서 `/api/export/md`를 호출합니다. - **cURL 예시:**

```
curl -H "Authorization: Bearer <ADMIN_TOKEN>" \
  -o "qualijournal_$(date +%Y%m%d)_AI.md" \
  "https://.../api/export/md?keyword=인공지능"
```

위 명령은 현재 날짜를 파일명에 포함하여, 해당 키워드 뉴스 Markdown을 파일로 저장합니다. (`-o` 옵션 사용) Authorization 헤더로 토큰을 포함해야 하며, 성공 시 Markdown 파일이 다운로드됩니다. 만약 파일명을 서버에서 Content-Disposition 헤더로 지정했다면 `-OJ` 옵션으로 원본 이름을 따를 수도 있습니다.

- **fetch 예시:** UI에서 "Markdown 내보내기" 버튼이 있는 경우

```
fetch(`/api/export/md?keyword=${currentKeyword}`, {
  headers: { 'Authorization': `Bearer ${token}` }
})
  .then(res => res.text())
  .then(mdContent => {
    console.log("Markdown content:", mdContent);
    // 1) 미리보기 모달에 표시하거나,
    // 2) Blob으로 변환해 파일 다운로드 트리거
    const blob = new Blob([mdContent], { type: 'text/markdown' });
    const url = URL.createObjectURL(blob);
    const a = document.createElement('a');
    a.href = url;
    a.download = `${currentKeyword}_${new Date().toISOString().slice(0,10)}.md`;
    document.body.appendChild(a);
    a.click();
    a.remove();
    URL.revokeObjectURL(url);
  });
```

이처럼 fetch로 Markdown 문자열을 받은 뒤, JavaScript로 Blob을 만들어 **클라이언트 측에서 다운로드**를 수행할 수 있습니다. 또는 단순히 `window.open('/api/export/md?...')`으로 새 탭/창 다운로드를 유도할 수도 있습니다.

Markdown 출력물 미리보기: 관리자 UI에서는 해당 Markdown을 HTML로 변환하여 **미리보기 모달**로 띄워줄 수 있습니다. 예를 들어, 서버가 `/api/export/md` 요청에 대한 HTML 미리보기 데이터를 주도록 할 수도 있고, 클라이언트에서 Markdown을 파싱하여 보여줄 수도 있습니다. 어쨌든, 이 기능을 통해 편집자는 최종 뉴스 페이지 모양을 발행 전에 검토할 수 있습니다 [24] [25].

`/api/export/csv` – CSV 내보내기

CSV 포맷은 **구조화된 데이터 표** 형태로 기사 목록을 저장합니다. 이 기능은 데이터 분석, 보고 용도 또는 백업을 위해 유용합니다. CSV에는 각 기사별 주요 필드가 컬럼으로 포함됩니다. 예를 들어: - Date(날짜), Keyword(키워드), Title(제목), Source(출처), Summary(요약문), URL(기사 원본 링크), Score(점수), Approved(승인여부), EditorNote(편집자 코멘트) 등.

사용 방법: - **cURL 예시:**

```
curl -H "Authorization: Bearer <TOKEN>" \
  -o "qualijournal_20251014_AI.csv" \
  "https://.../api/export/csv?keyword=인공지능"
```

이 명령은 해당 키워드의 기사 데이터를 CSV 파일로 저장합니다. 결과 파일은 Excel 등에서 열어볼 수 있습니다. 인코딩은 기본적으로 UTF-8이며, Excel 호환을 위해 BOM을 넣거나 필요시 EUC-KR로 변환해야 할 수도 있습니다 (한국어 깨짐 방지).

- **fetch 예시:** (UI에서 "CSV 다운로드" 버튼)

```
fetch(`/api/export/csv?keyword=${kw}`, { headers: { 'Authorization': `Bearer ${token}` } })
 .then(res => res.blob())
 .then(csvBlob => {
  const url = URL.createObjectURL(csvBlob);
  const a = document.createElement('a');
  a.href = url;
  a.download = `${kw}_${new Date().toISOString().slice(0,10)}.csv`;
  a.click();
  URL.revokeObjectURL(url);
 });
```

CSV의 Content-Type은 `text/csv`로 내려오므로 `res.blob()`으로 받아 파일로 저장하는 편이 적절합니다.

비고: - Export API들은 기본적으로 **편집 완료된 기사들(approved=True)**만을 대상으로 출력하도록 설계되었습니다. 만약 승인되지 않은 기사도 포함하려면 별도 옵션을 둘 수 있습니다. - 내보낸 Markdown 파일은 곧바로 독자 웹사이트나 블로그 등에 게시할 수도 있고, 아카이브용으로 축적됩니다. Cloud Run 백엔드에서도 `/api/export/md` 호출 시 **동시에 GCS에 MD/HTML/JSON 파일을 저장**하도록 구현하여, 자동 아카이브를 갱신할 수 있습니다 [26] . - CSV 파일은 주로 **내부 분석**이나 이후 DB 마이그레이션 시 데이터를 일괄 이전하는데 활용될 수 있습니다. (CSV를 불러와 DB에 import 등)

Export 기능을 통해 **운영 효율**과 **데이터 보존**이 크게 향상됩니다. 개발자는 이 API 구현 시 메모리에 있는 데이터 구조나 DB로부터 쿼리를 모아 정확한 포맷으로 직렬화해야 합니다. 운영자는 UI 버튼 하나로 손쉽게 결과물을 다운로드 받아 둘 수 있으며, 추후 문제가 생겨도 이 파일들을 참고해 **복구하거나 품질 개선**에 사용할 수 있습니다.

관리자 UI 개선 사항 및 사용 가이드

QualiJournal 관리자 UI는 편집자가 수집된 뉴스를 검토하고 **승인/편집/발행 작업을 수행하는 웹 인터페이스**입니다. 기존 JSON 편집 및 스크립트 실행 방식에서 발전하여, 보다 **직관적이고 편리한 UI/UX**를 제공합니다 [27] [28] . 이 섹션에서는 최신 관리자 UI의 주요 기능 개선과 사용법에 대해 설명합니다.

1. 관리자 토큰 입력 및 인증 유지

QualiJournal 관리자 UI는 **인증 토큰 기반 접근**을 사용합니다. 초기 로딩 시 ADMIN_TOKEN을 요구하여, 올바른 토큰을 입력해야만 API 호출이 가능하도록 설계되었습니다. 현재 구조에서는 입력한 토큰을 **브라우저 localStorage에 저장**하여 세션 동안 유지합니다.

관리자 UI 토큰 입력창 예시

- **토큰 입력창 필요 여부:** 현행 UI에서는 첫 방문 시 작은 모달이나 로그인 화면에서 토큰을 입력하도록 하고, 이후에는 localStorage에 저장된 토큰을 자동으로 불러와 사용합니다. 따라서 일반적인 **로그인 폼**처럼 한 번 입력하면 재방문 시에도 토큰이 유지되어 편집자가 반복 입력하지 않아도 됩니다. 별도의 토큰 입력창 메뉴는 두지 않았지만, 만약 토큰이 변경되거나 만료되었을 때를 대비해 **토큰 재설정 기능**이 필요할 수 있습니다. 예를 들어 설정 메뉴나 프로필 메뉴에 "토큰 다시 입력" 기능을 제공하여, 잘못된 토큰 사용 시 UI에서 갱신할 수 있게 해줍니다.
- **보안 고려:** 토큰을 localStorage에 보관하면 XSS 공격에 노출될 수 있으므로, 애플리케이션이 호스팅되는 도메인의 신뢰성을 확보하고, 필요시 쿠키 + HTTPOnly 등의 다른 방법을 검토해야 합니다. 현 단계에서는 간편히 localStorage를 사용하되, **토큰 입력 UI**를 통해서만 설정되게 하고, 콘솔을 통한 수동 조작 없이도 인증 절차를 완료할 수 있도록 했습니다.
- 운영자는 배포 시 설정한 ADMIN_TOKEN을 알고 있어야 하며, UI 처음 접속 시 제공된 필드에 입력합니다. 올바른 토큰일 경우 UI 상에서 "인증 성공" 메시지나 단순히 대시보드 화면으로 진입하며, 잘못 입력하면 **에러 알림**을 띄우고 재입력을 요구합니다.

요약하면, **토큰 입력창**은 사용자 친화성과 보안을 위해 **최초 1회는 필요**하며, 이후 세션 지속을 위해 localStorage(또는 cookie)에 저장해 활용합니다. 토큰이 변경될 경우 운영자에게 공유하여 새 토큰으로 재인증해야 함을 유념하세요.

2. 요약/번역 진행 상태 표시 (로딩 인디케이터)

뉴스 카드 요약 및 번역 작업은 수 초 이상의 시간이 걸릴 수 있으며, 사용자는 해당 작업이 이루어지고 있음을 알아야 합니다. 관리자 UI에는 **스피너(spinner) 또는 진행바**를 통해 요약/번역 요청이 처리 중임을 나타내는 기능이 도입되었습니다.

요약 작업 진행 중 표시 예시

- **요약/번역 버튼 상태:** 편집자가 "요약 생성" 또는 "번역" 관련 버튼을 클릭하면, 즉시 버튼을 비활성화(disabled)로 바꾸고 그 위나 옆에 로딩 아이콘을 표시합니다. 예를 들어 "요약 생성" 버튼 내부에 작은 원형 스피너가 돌아가거나, 버튼 텍스트를 "요약 중..."으로 일시 변경합니다.
- **전역 로딩 표시:** 만약 키워드 내 모든 기사에 대한 일괄 요약 (`/api/enrich/keyword`)을 진행하는 경우, 페이지 상단이나 중앙에 전역 로딩 스피너를 띄울 수 있습니다. 이렇게 하면 사용자가 작업이 진행되고 있음을 분명히 인지할 수 있습니다. 동시에, **다른 액션(새로고침 등)**을 하지 않도록 안내하거나, 취소가 어려움을 알리는 메시지를 함께 보여주는 것도 좋습니다.
- **완료 처리:** 서버 응답이 오면, 스피너를 제거하고 버튼을 원래 상태로 복귀시킵니다. 그리고 UI 데이터(기사 요약 필드 등)를 업데이트하여 사용자가 결과를 바로 확인할 수 있게 합니다. 예를 들어 요약문이 채워진 카드들이 화면에 표시되도록 합니다.
- **오류 시 표시:** 요청이 실패하면, 스피너를 제거하고 사용자에게 오류를 알려주는 모달이나 토스트 메시지를 띄웁니다. "요약 생성 실패: API 키 확인 필요" 등의 구체적인 원인을 표시하고, 사용자가 다음 행동(재시도 등)을 판단할 수 있게 합니다. 물론 오류 내용 중 민감한 정보(토큰 등)는 표시하지 않습니다.

이러한 진행 상태 표시 기능 덕분에 편집자는 **클릭 후 기다려야 하는 작업에서 불안감을 덜 수 있고**, 시스템이 동작 중이라는 피드백을 받게 됩니다. 개발자는 해당 UI 구현 시 **Promise 상태관리**나 **Vuex/Redux 상태**를 활용하여 `isLoading` 플래그를 두고 컴포넌트에서 조건부 렌더링으로 스피너를 표시하도록 했습니다. API 호출이 많은 관리자 도구 특성상, 일관된 로딩 UX를 주는 것이 중요합니다.

3. 발행 결과 미리보기 (Preview & Download)

뉴스 발행 전에 최종 결과물을 검토하거나 외부로 전달하기 위해, **미리보기 기능**과 **다운로드 링크** 제공이 추가되었습니다. 편집자는 UI에서 **"미리보기"** 버튼을 눌러 해당 키워드 뉴스 페이지가 실제로 어떻게 구성되는지 확인할 수 있습니다 [24].

발행 결과 미리보기 모달 예시

- **미리보기 모달:** 미리보기 버튼 클릭 시, `/api/export/md` API를 호출하여 Markdown 문자열을 받아온 뒤, 클라이언트에서 HTML로 변환하여 모달 창에 표시합니다. 모달에는 키워드 제목, 날짜, 기사 리스트가 **뉴스 웹페이지와 유사한 형태**로 렌더링됩니다. 편집자는 이 화면에서 잘못된 점(오탈자, 줄바꿈 문제, 등)을 발견하면 다시 UI 편집 화면으로 돌아가 수정할 수 있습니다. 미리보기는 실제 발행본과 동일한 Markdown/HTML을 사용하므로 신뢰할 수 있습니다.
- **다운로드 링크/버튼:** 모달 안이나 별도로 "Markdown 다운로드", "HTML 다운로드", "CSV 내보내기" 등의 버튼을 제공하여, 편집자가 원하면 **파일로 저장**할 수 있게 했습니다. 예를 들어 "Markdown 다운로드" 버튼은 `/api/export/md` 응답을 블롭(blob)으로 만들어 자동 다운로드를 트리거하며, "CSV 내보내기"는 `/api/export/csv`를 호출하여 파일을 내려받게 합니다 (전술한 fetch 코드 예시 참고). 이러한 내보내기 버튼을 통해 편집자가 결과물을 이메일로 전송하거나 아카이브 폴더에 복사해 둘 수 있습니다.
- **툴팁:** UI 내 중요한 버튼들(예: 발행, 미리보기, 내보내기 등)에는 **툴팁**을 달아 기능 설명을 제공했습니다. 예를 들어 미리보기 버튼에 마우스를 올리면 "발행될 페이지 미리보기"라는 툴팁이 나타나도록 하여, 처음 쓰는 사람도 헷갈리지 않게 합니다. 툴팁 구현은 간단한 title 속성 또는 UI 라이브러리 컴포넌트를 활용했습니다.

발행 실행: QualiJournal 관리자 UI에서는 최종 검토 후 **발행(Publish)** 단계를 실행할 수 있습니다. 만약 시스템이 `/api/publish`와 같은 엔드포인트를 가진다면, 발행 버튼 클릭 시 해당 API를 호출하여 서버 측에서 **아카이브 파일 저장 및 최종 확정**을 처리합니다 [29]. 현 구현에서는 별도의 publish API 대신, **Export (MD/HTML)**를 수행하면 사실상 발행과 동일하게 간주하고, 그 결과물을 수동으로 게시하는 형식일 수 있습니다. 운영 정책에 따라: - Cloud Run 백엔드에서 export 시 **자동으로 archive 버킷에 저장**하게 했으면, UI에서 굳이 publish 액션이 필요 없고 export == publish입니다. - 아니면 UI에서 "최종 발행" 버튼을 만들어 `/api/publish`를 호출하게 할 수 있는데, 이 경우 서버는 archive 저장 후 성공 응답을 보내고, UI는 "발행 완료" 메시지를 보여줍니다.

편집자는 **미리보기 확인 → 발행/내보내기 실행** 순서로 하루 작업을 마무리하게 됩니다. UI 개선을 통해 이제 JSON 파일을 직접 편집하던 번거로움 없이, 웹 화면에서 클릭 몇 번으로 승인 및 발행이 이뤄집니다. 또한 검색/필터, 타입별 보기, 키보드 단축키 등 편의 기능이 더해져 작업 효율을 높였습니다 [28] [30]. 몇 가지 예를 들면: - **키워드 입력 및 기록:** 상단에 키워드 입력 창이 있어 새로운 키워드 수집을 바로 시작할 수 있고, 이전에 발행했던 키워드들은 자동 완성이나 드롭다운으로 조회할 수 있습니다. - **유형별 필터링:** 뉴스/논문/커뮤니티 등 **콘텐츠 유형별 탭**이나 체크박스를 제공하여, 보고 싶은 카테고리만 리스트에 보이도록 할 수 있습니다 [31]. - **정렬 및 점수 표시:** 각 기사 카드에는 **신뢰도 점수(QG/FC)**나 키워드 적합 점수가 아이콘/배지로 시각화되어 있어, 편집자가 우선순위를 판단하기 쉽습니다 [32]. - **승인/보류/제외 액션:** 기사 카드별로 "✔ 승인", "✖ 제외", "보류" 버튼이 있고, 클릭 시 즉시 상태가 반영됩니다 (UI 상에서 색상이나 필터로 표시). 승인된 항목만 나중에 발행에 포함됩니다 [28]. 잘못 눌렀을 경우 취소(undo)도 가능하게 할 수 있습니다. - **편집자 코멘트 입력:** 각 카드에는 편집자가 한 줄 의견을 남길 수 있는 입력란이 있습니다 [33] [22]. 기본으로 자동 생성된 요약문이나 기사 주요 문장을 넣어 두고, 편집자가 이를 수정하거나 덧붙이는 방식입니다. 이 코멘트는 최종 뉴스 페이지에 편집장 한마디처럼 표시될 수 있습니다.

위 개선 사항들이 반영된 **관리자 UI** 덕분에, 운영자는 기술적인 부분에 신경쓰지 않고도 콘텐츠에 집중하여 하루 뉴스 발행 작업을 수행할 수 있습니다. 개발자는 이러한 UI 기능들이 원활히 동작하도록 백엔드 API와 실시간 상호작용(예: 승인 시 데이터 상태 변화)을 구현해야 합니다. 필요시 WebSocket이나 SSE로 여러 편집자 간 실시간 동기화도 고려할 수 있습니다 [34].

(스크린샷 및 예시는 실제 UI의 모습을 추정하여 작성한 것입니다. 운영 환경에 따라 UI 구성은 달라질 수 있으나, 핵심 개념은 동일합니다.)

자동화 테스트 루틴 (pytest + mock 활용)

시스템의 안정성을 보장하기 위해서는 **자동화 테스트**가 필수입니다. QualiJournal 프로젝트는 `pytest` 프레임워크를 사용하여 단위 테스트와 통합 테스트를 구현하며, 외부 의존은 `mock`을 통해 격리합니다 [35]. 이 섹션에서는 테스트 구조와 실행 방법, 주요 테스트 시나리오를 설명합니다.

1. 테스트 구조 설계

테스트 코드는 프로젝트 내 `tests/` 디렉토리에 위치하며, 모듈별로 세부 디렉토리 또는 파일로 구성됩니다: - `tests/test_collect.py`: **수집 모듈 테스트** – 여러 출처로부터 데이터를 올바르게 수집하는지 검증. 크롤러 함수에 대해 **모의 웹 응답**(requests를 mock)으로 RSS 피드나 HTML을 흉내내 desired output이 나오는지 테스트합니다. - `tests/test_preprocess.py` (또는 통합): **재구성 및 필터링 테스트** – 중복 제거, 품질게이트(QG) 적용, 점수 계산 로직 테스트. 다양한 기사 메타데이터를 입력으로 주고 결과 필터링이 기대대로 동작하는지 확인합니다. - `tests/test_summary.py`: **요약/번역 모듈 테스트** – OpenAI나 DeepL API 호출 부분을 **mock으로 대체**하여, 예를 들어 항상 동일한 번역 결과를 반환하도록 한 뒤 요약 함수가 결과 문자열을 적절히 파싱/저장하는지 확인합니다. API 키 없이도 테스트 가능하도록 `responses` 라이브러리나 `unittest.mock.patch`로 외부 요청을 가로챕니다. - `tests/test_publish.py`: **발행/출력 테스트** – Markdown 생성 함수 테스트. 몇 개의 기사 데이터를 입력해 Markdown 문자열을 만들었을 때 헤더, 본문, 링크 형식이 올바른지 검사합니다. 혹은 CSV 생성 기능도 dummy 데이터를 넣어 CSV 행 수와 컬럼을 검증합니다. - `tests/test_api.py`: **API 레이어 테스트** – FastAPI나 Flask의 테스트 클라이언트를 이용하여 `/api/report`, `/api/enrich/...`, `/api/export/...` 엔드포인트를 호출해 보는 통합 테스트를 작성합니다. 여기서는 위 모듈들의 기능이 조합되어 동작하므로, e2e에 가깝게 검증합니다. ADMIN_TOKEN 검증을 위해 테스트 시에는 미리 환경변수에 토큰을 세팅해두고, 헤더에 넣어 요청 보내 200 응답을 받는지 확인합니다. 또한 잘못된 토큰으로 401이 나오는지, 누락 파라미터로 400을 리턴하는지 등 **예외 케이스**도 다룹니다.

테스트 코드에서는 **Arrange-Act-Assert** 패턴을 준수하여 가독성을 높입니다:

```
# Arrange: 주어진 환경 설정
test_client = app.test_client()
os.environ["ADMIN_TOKEN"] = "testtoken123"

# Act: 액션 수행
resp = test_client.post("/api/report?keyword=테스트", headers={"Authorization": "Bearer
testtoken123"})

# Assert: 결과 확인
assert resp.status_code == 200
data = resp.get_json()
assert "articles" in data
assert len(data["articles"]) > 0
```

또한 **fixture**를 활용해 반복되는 세팅 (예: 더미 기사 리스트 생성 함수, 임시 파일 환경 등)을 정리하고, **parametrize**를 통해 다양한 시나리오를 커버합니다.

2. Mocking 및 테스트 격리

일부 테스트는 외부 시스템 의존성이 크므로, `pytest-mock`이나 표준 `unittest.mock`을 적극 활용합니다: - **외부 API 모킹:** 예를 들어 `summarize_text()` 함수 내에서 `openai.Completion.create(...)`를 호출한다면, 테스트에서는 `with patch('openai.Completion.create') as mock_api:`로 감싸 해당 함수가 가짜 응답을 리턴하도록 합니다. 이렇게 하면 **요금이 발생하지 않고** 예측 가능한 결과로 테스트할 수 있습니다. - **DB나 GCS 모킹:** 만약 DB_URL이 설정되어 DB 연동을 한다면, 테스트 환경에서는 SQLite 메모리 DB로 교체하거나, DB accessor 함수를 mock 처리하여 실제로 DB 쓰기는 안 하도록 해야 합니다. GCS 연동 부분도 `storage.Client().bucket(...).blob(...).upload_from_string()` 등 메소드를 patch하여 실제 GCP 호출 없이도 동작을 가장합니다. - **파일 시스템 격리:** 앱이 로컬 JSON 파일을 다룬다면, 테스트 시작 시 임시 디렉

토리를 만들어 해당 경로를 config로 쓰게 하거나, 테스트 완료 후 파일을 삭제하도록 합니다. Pytest의 `tmp_path` fixture를 이용해 경로를 격리할 수 있습니다.

예시: OpenAI API 모킹

```
from unittest.mock import patch

def test_summary_generation(monkeypatch):
    # 가상 응답 생성
    fake_response = {"choices": [{"text": "요약 결과 문장."}]}
    # openai.Completion.create 호출시 fake_response 반환
    monkeypatch.setattr("openai.Completion.create", lambda **kwargs: fake_response)
    # 대상 함수 실행
    result = summarize_text("This is a long English text...")
    assert "요약 결과 문장." in result
```

위처럼 monkeypatch 또는 patch를 사용하면, 테스트 중에는 외부 API가 아닌 가짜 결과를 사용하므로 안정적 테스트가 가능합니다.

3. 테스트 실행 및 커버리지

- **로컬 실행:** 개발자는 로컬에서 `pytest` 명령으로 전체 테스트를 실행할 수 있습니다. `pytest -v`로 자세한 로그와 함께 실행하고, `-s` 옵션으로 print 출력도 볼 수 있습니다. 모든 테스트가 통과해야만 코드를 배포하거나 머지하도록 기준을 세웁니다.
- **커버리지 측정:** `pytest-cov`를 사용해 코드 커버리지를 측정합니다. 예: `pytest --cov=qualijournal --cov-report=term-missing` 명령으로 각 모듈 별 몇 % 테스트되었는지 출력합니다. 핵심 로직(수집, 요약, 발행 등)은 가급적 높은 커버리지(> 90%)를 유지하도록 작성합니다. UI 코드는 별도로 프론트엔드 테스트(Vue/React의 Jest 등)를 둘 수 있지만, 여기서는 백엔드 위주로 설명합니다.
- **CI 통합:** CI/CD 파이프라인에서 배포 전에 자동으로 `pytest`를 실행하도록 구성합니다. (다음 CI/CD 섹션 참고) 테스트 실패 시 배포가 중단되므로, 항상 테스트 코드를 최신화하고 새로운 기능 추가 시 테스트도 함께 작성합니다.

4. 반복 실행 루틴 테스트

QualiJournal의 **일일 발행 루틴** (collect → enrich → export)도 통합 테스트로 검증합니다 [35] . 예컨대 가상의 키워드 "TEST"에 대해: 1. `/api/report?keyword=TEST` 호출 -> 200 OK와 article list 반환 확인. 2. `/api/enrich/keyword?keyword=TEST` 호출 -> 요약 필드 채워지는지 확인. 3. 일부 기사 approval 필드 토글 -> (테스트에선 수동으로 data 조작 or API 호출로 simulate) 4. `/api/export/md?keyword=TEST` 호출 -> 반환된 Markdown에 approved 기사들의 제목이 모두 들어있는지, 형식이 맞는지 검사.

이러한 흐름을 자동화해두면, 개발자가 변경을 가했을 때 핵심 시나리오가 깨지지 않았는지 빠르게 알 수 있습니다. 특히 일정 기반 자동화나 멀티스레드 환경에서는 **동시성 이슈**가 없는지도 테스트해야 합니다. pytest로는 복잡하지만, 가능하면 스케줄링을 가상 시뮬레이션하거나, 같은 API를 한 번에 여러 번 호출해 보는 식으로 검사할 수 있습니다.

결론적으로, 철저한 자동화 테스트는 QualiJournal 시스템을 **안정적이고 신뢰성 있게 발전**시키는 토대입니다. 개발자는 새로운 기능 추가 시 반드시 대응하는 테스트를 추가하고, 운영자는 배포 전에 테스트가 통과했는지 확인함으로써 품질을 담보할 수 있습니다.

(pytest와 mock 코드는 예시이며 실제 구현과 다를 수 있으나, 개념적으로 해당 방향으로 테스트를 구성해야 함을 보여줍니다.)

CI/CD 자동 배포 구성 (GitHub Actions 기반)

지속적 통합과 배포(CI/CD)를 도입하면 코드를 손쉽게 배포하고 버전을 관리할 수 있습니다. 여기서는 **GitHub Actions**를 이용한 Cloud Run 자동 배포 파이프라인을 설정하는 방법을 설명합니다 (Google Cloud Build를 사용할 수도 있으나, Actions로 GitHub과의 연계를 보여줍니다).

1. GitHub Actions 워크플로우 설정

GitHub 리포지토리에 `.github/workflows/deploy.yml` 파일을 생성하여 CI/CD 파이프라인을 정의합니다. 예시 Workflow:

```
name: CI and Deploy

on:
  push:
    branches: [ "main" ]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    env:
      PROJECT_ID: my-gcp-project-id
      REGION: asia-northeast3
      SERVICE: qualijournal-service
    steps:
      - name: Checkout source
        uses: actions/checkout@v3

      - name: Set up Google Cloud SDK
        uses: google-github-actions/setup-gcloud@v1
        with:
          project_id: ${{ env.PROJECT_ID }}
          service_account_key: ${{ secrets.GCP_SA_KEY }}
          export_default_credentials: true

      - name: Configure Docker auth
        run: gcloud auth configure-docker asia-northeast3-docker.pkg.dev

      - name: Build and Push image
        run: |
          docker build -t asia-northeast3-docker.pkg.dev/${{ env.PROJECT_ID }}/containers/$
{{ env.SERVICE }}:${{ github.sha }} .
          docker push asia-northeast3-docker.pkg.dev/${{ env.PROJECT_ID }}/containers/$
{{ env.SERVICE }}:${{ github.sha }}

      - name: Deploy to Cloud Run
        run: |
```

```
        gcloud run deploy ${{ env.SERVICE }} \
          --image asia-northeast3-docker.pkg.dev/${{ env.PROJECT_ID }}/containers/${{ env.SERVICE }}:${
{ github.sha }} \
          --region ${{ env.REGION }} \
          --platform managed \
          --allow-unauthenticated \
          --update-env-vars ADMIN_TOKEN=${{ secrets.ADMIN_TOKEN}},API_KEY=${
{ secrets.API_KEY}},DB_URL=${{ secrets.DB_URL}},GCS_BUCKET=${{ secrets.GCS_BUCKET}}
```

위 YAML은 다음을 수행합니다: - main 브랜치에 push될 때 트리거됩니다. - **Checkout**: 코드를 가져옵니다. - **GCloud 세팅**: GCP 서비스 계정 JSON 키를 `secrets.GCP_SA_KEY`에서 불러와 인증합니다. (사전에 GitHub 리포지토리에 이 키를 저장하고, 권한은 Cloud Run Admin, Artifact Registry Write 등을 줘야 함) - **Docker 인증**: Artifact Registry에 Docker 푸시할 수 있도록 도커 인증 설정. - **Build & Push**: 도커 이미지를 빌드하고, **커밋 SHA 태그**로 Artifact Registry에 푸시. (이로써 이미지가 고유 태그로 저장되고, rollback 등이 용이) - **Deploy**: Cloud Run 서비스에 해당 이미지를 배포. secrets에 저장된 ADMIN_TOKEN 등 민감정보를 `--update-env-vars`로 주입합니다. (secret을 GitHub에 넣을 때도 암호화돼 있지만, 가능하면 GCP Secret Manager로 옮기는게 더 안전하나, 여기선 편의상 GH secret 사용)

Secrets 설정: 위 workflow가 제대로 동작하려면 GitHub 리포지토리 Settings의 Secrets에 GCP_SA_KEY (JSON), ADMIN_TOKEN, API_KEY, DB_URL, GCS_BUCKET 값을 등록해야 합니다. 운영자는 이러한 값을 관리하며 변경 시 secrets도 업데이트해야 합니다.

2. CI 단계 – 테스트 수행

배포 전에 **테스트를 자동으로 실행**하여 품질을 보증합니다. GitHub Actions 워크플로우에 테스트 단계를 추가할 수 있습니다:

```
    - name: Run Tests
      run: |
        python -m pip install -r requirements.txt
        python -m pytest
```

테스트가 실패하면 워크플로우는 중단되고, 배포가 진행되지 않습니다. 이를 통해 **테스트 통과된 코드만 프로덕션에 배포**됩니다.

3. Cloud Build 대안

만약 GitHub Actions 대신 **Cloud Build**로 CI/CD를 구성하려면, GCP에 **Cloud Build 트리거**를 설정합니다: - 저장소를 GCP에 연결하고 (GitHub 또는 Cloud Source Repositories), - 특정 브랜치 push 시 cloudbuild.yaml을 실행하도록 합니다. - `cloudbuild.yaml`에서는 steps로 `docker build` → `docker push` → `gcloud run deploy`를 지정하면 됩니다. - Secret Manager 값을 substituion으로 받아와 deploy 명령에 포함시키거나, Cloud Run 서비스에 미리 설정해두고 `--update-env-vars`는 생략할 수도 있습니다.

둘 중 어떤 방식을 쓰든, 핵심은 **코드 변경 → 자동 테스트 → 컨테이너 이미지 빌드/배포** 순으로 이루어지게 하는 것입니다 [2]. 이를 통해 사람의 실수 없이도 항상 최신 코드가 배포되고, 문제가 발생하면 버전 추적 및 롤백이 쉽게 됩니다.

4. 배포 후 모니터링

CI/CD 파이프라인이 성공하면: - GitHub Actions의 경우 **Actions 탭**에서 성공 로그를 확인할 수 있고, Cloud Run 콘솔에서 새로운 리비전이 트래픽을 받고 있는지 확인합니다. - Cloud Build의 경우 **Cloud Build History**에서 빌드 로그를 검토합니다. - 배포된 서비스는 **Cloud Monitoring**이나 자체 헬스체크를 통해 확인합니다. 예: `/api/health` 엔드포인트를 만들어 두었다면, Actions 마지막에 `curl`로 호출해 200 OK인지 확인하는 step을 추가할 수도 있습니다.

5. 운영자 측면 주의사항

운영자는 CI/CD를 통해 **배포 자동화**가 되었더라도, 배포 시점과 내용을 파악하고 있어야 합니다. 예를 들어: - 새로운 기능이 배포되면 그에 대한 가이드(이 문서)가 업데이트되었는지 확인하고, 필요한 경우 편집자에게 UI 변경점을 공유합니다. - 배포 후 첫 실행 시 혹시 모르는 이슈가 나타나면 재빨리 이전 버전으로 롤백하거나 hotfix 워크플로우를 밟습니다. GitHub Actions에서 수동 트리거(workflow dispatch)로 특정 이전 SHA를 배포하도록 구성할 수도 있습니다.

CI/CD의 도입으로 개발자는 코드 작성에 집중하고, 운영자는 **항상 테스트 통과된 코드**만 접하게 되어 신뢰도가 높아집니다. QualiJournal 시스템은 이러한 자동 배포 체계를 통해 **지속적으로 개선 및 고도화**되면서도 일일 운영에 지장을 주지 않는 안정성을 추구합니다.

보안 및 로깅, GCS 연동 백업 방법

운영 환경에서 시스템을 안정적으로 유지하려면 **보안 강화**, **로그 모니터링**, **백업 체계**가 중요합니다. QualiJournal 시스템은 비교적 단순한 구조이지만, 아래의 방안들을 적용하여 운영 리스크를 줄입니다.

1. 보안 (Security)

- **API 인증 및 접근제어:** 모든 관리자 API는 ADMIN_TOKEN으로 보호되고 있으나, 추가로 **IP 제한**이나 **Cloud Run IAM** 권한으로 2중 보호를 고려할 수 있습니다. 예를 들어 Cloud Run 서비스를 **인증 필요**로 설정하고, 운영자에게 GCP IAM 계정을 발급한 뒤, IAP(Identity Aware Proxy) 등을 통해 로그인한 사용자만 UI 접근하게 하는 방법도 있습니다. 다만 현재는 토큰 기반으로 충분히 커버되므로, UI URL이 유출되지 않도록 하고 토큰 관리에 주력합니다.
- **데이터 전송 암호화:** Cloud Run 제공 URL은 기본적으로 HTTPS(SSL)입니다. Custom domain을 사용하는 경우에도 SSL 인증서를 설정하여 (예: Cloud Run 도메인 매핑 + Cloud DNS + 인증서) **모든 트래픽을 HTTPS로 유지**해야 합니다. 이렇게 하면 ADMIN_TOKEN이나 API_KEY 등이 네트워크 상에서 노출될 위험을 낮춥니다.
- **민감 정보 취급:** 로그나 에러 메시지에 **비밀번호, 토큰, 키** 등의 값이 출력되지 않도록 주의합니다. 예를 들어 예외 발생 시 `.env` 내용을 통째로 출력한다든지 하지 말고, 필요한 부분만 로깅하고 나머지는 숨깁니다. 또한 GitHub 등 소스 저장소에 .env를 올리지 않고, 문서에서도 키 값을 마스킹합니다.
- **의도치 않은 공개 차단:** Cloud Run 서비스를 `--allow-unauthenticated`로 둔 경우, 이 URL은 인터넷 누구나 접근 가능합니다. ADMIN_TOKEN 없으면 핵심 기능은 막히지만, 혹시 public GET 엔드포인트가 있다면 악용될 수 있습니다. 필요시 Cloud Run 설정에서 "인증된 호출자만" 옵션으로 바꾸고 Cloud Scheduler나 UI에는 별도 인증 구현을 해야 합니다. (이 부분은 운영 편의 vs 보안의 트레이드오프이므로 프로젝트 상황에 맞게 결정)
- **취약점 업데이트:** 주기적으로 사용하는 **라이브러리 의존성을 업데이트**하고, 알려진 취약점(CVE)이 있는 패키지는 신속히 패치합니다. Docker 베이스 이미지도 (예: python:3.x slim) 정기적으로 갱신하여 보안 패치가 적용되도록 합니다.
- **관리자 권한 관리:** 만약 복수의 편집자가 있을 경우, ADMIN_TOKEN을 여러 개 발급하여 (예: JWT 같은 것 고려) 각자 개인 토큰을 쓰게 하고, 혹시 토큰 유출시 그 토큰만 폐기하는 방법도 생각해볼 수 있습니다. 현재는 단일 토큰 체계지만, 장기적으로 **로그인 시스템**으로 확장 여지도 있습니다.

2. 로깅 및 모니터링

- **Cloud Logging 활용:** Cloud Run에서 출력하는 모든 표준 출력/표준 에러 로그는 GCP Cloud Logging에 자동 수집됩니다. 개발 시 `logging` 라이브러리를 사용하여 **구조화 로그** (JSON 형태)로 기록하면, Cloud Logging에서 필터링과 검색이 용이합니다. 예:

```
logging.info({"event": "collect_start", "keyword": kw, "user": user_id})
```

  이런 로그는 JSON 필드별 필터링이 가능하며, 오류 시 stacktrace도 함께 기록하도록 합니다.
- **로그 모니터링:** GCP Operations > Logging 에서 **로그 탐색기**를 사용해 특정 키워드(예: ERROR, /api/report, keyword 이름 등)로 검색하여 문제 상황을 파악합니다. **경고/오류 레벨 로그에 알림**을 붙일 수도 있습니다. (예: Cloud Monitoring에서 로그 기반 경고 설정하여 ERROR 로그 발생 시 Slack이나 이메일 알람)
- **예외 처리와 로깅:** 코드에서 예상치 못한 예외 발생을 최대한 캐치하여 **사용자에게는 친절한 메시지, 개발자용 로그에는 상세 스택**을 남기도록 합니다. Flask나 FastAPI의 exception handler를 사용해 500 오류를 잡고 로그를 찍는 식입니다. 예를 들어 `/api/report`에서 키워드가 누락되면 400을 리턴하고 `"Missing keyword parameter"`라고 응답하되, 로그에는 호출한 사용자 IP와 함께 해당 에러를 남겨 원인 파악에 도움을 줍니다.
- **애플리케이션 메트릭:** 필요하면 수집 기사 수, 요약 API 호출 횟수, 평균 처리시간 등의 메트릭을 수집/시각화합니다. Cloud Run은 요청 횟수, 레이턴시 등의 기본 메트릭을 제공하므로, GCP Monitoring 대시보드에서 확인할 수 있습니다. CPU/메모리 사용량을 모니터링하여 인스턴스 크기 조정을 검토합니다.

3. GCS 연동 및 백업

QualiJournal는 **발행 결과와 원본 데이터를 백업**하기 위해 Google Cloud Storage(GCS)를 사용합니다. GCS를 통해 Cloud Run의 비영구적 스토리지 한계를 해결하고, 장기 보존 및 복원력을 확보합니다 [36].

- **발행물 아카이브 저장:** 뉴스가 발행될 때마다 (혹은 `/api/export/md` 실행 시) **Markdown, HTML, JSON 파일**을 GCS 버킷에 저장합니다. 예를 들어 버킷 이름 `qualijournal-archive` 아래에 `2025-10-14_인공지능.md`와 동일 내용의 `.html`, `.json`을 업로드합니다 [26]. JSON에는 `date`와 `articles` 배열이 포함된 구조로, 해당 키워드의 모든 기사와 메타데이터(요약, 코멘트, 점수 등)가 보존됩니다. 이렇게 저장된 아카이브는 나중에 **독자용 웹사이트**에 노출하거나, 문제가 발생했을 때 **재발행/데이터 분석**에 활용할 수 있습니다.
- **첨부자료 백업:** 만약 뉴스 카드에 이미지나 첨부파일이 있다면 (현재 시나리오에는 없지만 향후 추가 가능), GCS에 같이 저장하거나 별도 버킷에 관리합니다. GCS Object 이름은 날짜+키워드 등으로 네이밍 규칙을 정해두면 찾기 쉽습니다.
- **DB 스냅샷/마이그레이션 백업:** 현재는 DB를 사용하지 않더라도, 추후 DB 도입 시 **주기적인 DB 백업**을 GCS에 저장해야 합니다. 예를 들어 Cloud SQL을 쓰면 자동백업이 있으나, NoSQL(Firestore 등)이면 별도 export 기능을 써야 합니다. CSV export 기능으로 어느 정도 백업 대체를 하지만, 스키마 변화 등에는 한계가 있으므로 공식 백업 기능을 이용합니다.
- **백업 주기 및 관리:** 아카이브 파일은 매일 생성되므로, 용량 관리가 필요합니다 [37]. GCS에서는 Lifecycle 규칙을 설정해 일정 기간(예: 1년) 지난 객체를 삭제하거나 Nearline/Coldline으로 자동 전환해 비용을 절감할 수 있습니다. 운영자는 스토리지 용량과 비용을 모니터링하여 필요시 규칙을 적용합니다. 중요한 데이터는 삭제 전에 별도 오프라인 저장해둘 수도 있습니다.
- **복구 방법:** 만약 최신 발행본이 손상되거나 시스템 장애가 발생하면, GCS에 있는 최근 Markdown 파일을 수동으로 내려받아 웹에 게시하거나, JSON 데이터로부터 다른 시스템에서 렌더링해 볼 수 있습니다. GCS는 **99.999999999% 내구성**을 제공하므로, 백업 저장소로 신뢰할 수 있습니다.

4. 추가 권고 사항 (보안/백업 관련)

- **권한 분리:** 운영자(편집자)에게는 필요한 최소한의 GCP 권한만 부여합니다. 예를 들어, 편집자는 Cloud Run 서비스 URL로 UI에 접근만 하면 되므로 GCP Console 접근은 불필요합니다. 만약 접근 필요시에도 **보기 권한** 정도로 최소화합니다. 개발자에게는 서비스 수정 권한, 운영자에겐 로그 뷰어 권한 정도로 나누는 것도 방법입니다.
- **테스트 환경 분리:** 운영용 프로젝트와 별도로 개발/스테이징 GCP 프로젝트를 두고, 테스트용 Cloud Run, GCS 버킷을 운용합니다. 보안 키도 각기 다르게 해서, 테스트 환경 노출이 곧 운영 침해로 이어지지 않도록 합니다.
- **정기 점검:** 보안패치, 키 교체, 백업 유효성 등을 **정기 점검** 일정에 넣습니다. 아래 운영 루틴 섹션에 언급할 아침 점검 시, 전일 백업 파일이 잘 올라갔는지, 로그에 이상 징후는 없는지 함께 확인하도록 합니다.

종합하면, QualiJournal 시스템은 **간결한 구조 속에서도 강건한 보안과 데이터 보존**을 목표로 합니다. 운영자는 위 방안들을 참고하여 시스템이 장기간 안전하게 서비스될 수 있도록 주기적으로 관리해야 합니다.

향후 DB 전환 및 마이그레이션 설계 요약

현재 QualiJournal은 기사 데이터를 JSON 파일로 저장/관리하고 있으나 [38] [39], 향후 **데이터베이스(DB)**로 전환하여 확장성과 협업 기능을 높이는 것이 고려되고 있습니다. 이 섹션에서는 미래에 예상되는 DB 도입 방향과 기존 데이터 마이그레이션에 대한 간략한 설계를 정리합니다.

1. 왜 DB로 전환하나?

- **데이터 일관성과 동시성:** JSON 파일은 단순하고 초기 구현이 쉬웠지만, 여러 사용자가 동시에 편집하거나, 데이터가 많아질 경우 경합 및 성능 문제가 생길 수 있습니다. DB는 트랜잭션과 잠금 기능으로 동시에 여러 편집자가 작업해도 데이터 정합성을 유지합니다 [34].
- **질의 및 역사 관리:** DB를 사용하면 특정 키워드의 과거 기사들을 **쿼리로 쉽게 조회**하거나 통계를 낼 수 있습니다. JSON 파일은 개별 파일을 일일이 파싱해야 하지만, DB에서는 SQL이나 인덱스를 통해 **키워드별 히스토리**나 **검색 기능**을 구현하기 용이합니다.
- **확장 및 통합:** 앞으로 **독자용 인터페이스**나 다른 시스템과 연동시, 표준 DB에 접근하는 것이 더 수월합니다 [40]. 또한 새로운 기능 (예: 사용자 피드백 저장, 추천 알고리즘 기록 등)을 추가하려면 구조화된 저장소가 적합합니다.

2. DB 종류와 구조 설계

- **DB 종류 선택:** 고려되는 옵션은 **관계형 DB**(예: PostgreSQL, MySQL) 또는 **NoSQL**(예: Cloud Firestore, MongoDB)입니다.
- 관계형 DB의 장점: 복잡한 조인이나 스키마 강제, ACID 트랜잭션. 예를 들어 기사 테이블, 키워드 테이블을 분리하고 관계를 맺으면 데이터 중복을 줄이고 **정규화**할 수 있습니다.
- NoSQL의 장점: JSON 구조 저장에 친화적이고 스키마 유연성, 초당 높은 쓰기 처리. 현재 JSON 구조를 거의 그대로 저장하고, 키워드별 도큐먼트로 관리하는 Firestore 방식은 마이그레이션이 비교적 쉽습니다.
- **스키마 제안 (관계형 가정):**
- `keywords` 테이블: `id (PK)`, `keyword` (키워드 텍스트), `date` (발행일), 기타 메타 (예: 편집자, 발행여부).
- `articles` 테이블: `id (PK)`, `keyword_id (FK to keywords)`, `title`, `source`, `pub_date` (기사 원문 일자), `summary`, `url`, `score`, `approved` (bool), `editor_note`, `type` (뉴스/논문 등 카테고리), `lang` (언어) 등 컬럼. 키워드 하루치 기사 여러 개가 이 테이블에 행으로 연결됩니다.
- 추가로 `sources` 테이블 (출처별 신뢰도 점수 등 저장)이나 `editors` 테이블 (편집자 계정 관리) 등을 둘 수 있지만, 초기에는 단순하게 갑니다.
- **스키마 제안 (NoSQL 가정 - Firestore):**

- 컬렉션 `daily_news` 아래에 도큐먼트 `YYYY-MM-DD_<키워드>` 이름으로 저장. 각 도큐먼트 필드로 `keyword`, `date`, `articles` (배열) 등이 포함. JSON 파일과 거의 동일한 구조로 저장되나, Firestore 인덱스를 통해 날짜/키워드별 조회를 빠르게 할 수 있습니다.
- 또는 컬렉션을 `keywords`로 하고 하위컬렉션에 articles를 넣는 방식을 쓸 수도 있습니다.
- Firestore는 실시간 리스너로 데이터 변경을 UI에 즉각 반영하는데도 활용 가능하여, 여러 편집자가 함께 작업할 경우 유용합니다 (단, 작업량이 많으면 요금 고려).

3. 마이그레이션 계획

- **데이터 이관 절차:** 기존 아카이브 JSON 파일 또는 `selected_keyword_articles.json` 등의 데이터를 DB로 옮겨야 합니다. 이를 위한 **마이그레이션 스크립트**를 작성합니다. Python으로 작성 시, DB client (psycopg2 또는 firestore admin SDK 등)를 사용하여:
- GCS나 로컬의 기존 JSON 파일을 모두 읽어들입니다. (혹은, 하루씩 선택해서 점진적으로)
- 각 JSON의 내용을 위에서 정의한 DB 스키마에 맞게 INSERT합니다. 관계형 DB라면 키워드별 ID 매핑을 관리하고, 기사 하나마다 INSERT. NoSQL이면 도큐먼트를 만들어 batch write.
- 데이터 양이 많다면 pagination 처리나 병렬 처리를 고려하지만, 하루 한 키워드씩이라 크기는 manageable 할 것입니다.
- 검증: 임의의 하루를 골라 JSON과 DB 내용을 비교하여 손실 없이 옮겨졌는지 확인합니다.
- **이중 운영 기간:** 마이그레이션 완료 후에도, 일정 기간 JSON 파일 방식과 DB 방식을 **병행 운영**할 수 있습니다. 즉, API 호출 시 두 군데 다 쓰고, 읽기는 DB 우선하되 JSON도 쓰는 방식으로 해두고, 문제 없으면 JSON 경로를 제거하는 방식입니다. 또는 가장 최근 N일만 DB에서 관리하고 나머지는 archive에서 on-demand로 읽도록 할 수도 있습니다.
- **기능 수정:** DB로 전환하면 API 내부 구현을 JSON 파일 처리에서 DB 쿼리로 변경해야 합니다 [41]. 예를 들어 `/api/keyword/{kw}` (보고서 조회)는 DB에서 SELECT * FROM articles WHERE keyword=...로 가져오도록 수정. 또한 편집자 승인 등의 기능은 DB update문으로, 발행 export는 JOIN이나 SELECT 결과를 Markdown으로 변환하도록 변경합니다. 이때 ORM을 도입하면 (SQLAlchemy 등) 생산성을 높일 수 있습니다.
- **성능 고려:** DB 도입 후 적절한 인덱스를 설정해야 합니다. 키워드+날짜 조합에 인덱스를 생성하여 빠르게 해당 키워드의 기사들을 가져올 수 있게 합니다. NoSQL인 경우, Firestore는 기본 키 기반이 문서명이라, 날짜/키워드를 문서명으로 하거나, 별도의 필드 인덱스 세팅이 필요합니다.
- **트랜잭션:** 승인/발행처럼 **여러 상태를 한번에 변경**하는 경우, 관계형 DB는 트랜잭션으로 처리합니다. 예를 들어 15개 기사를 일괄 승인 -> 트랜잭션 커밋. Firestore도 batch write나 transaction API가 있으므로 활용합니다. 이를 통해 데이터 상태 불일치(일부는 변경되고 일부는 실패 등)를 막습니다.
- **Migration 도구:** 관계형의 경우 Alembic 등의 마이그레이션 툴로 스키마 버전을 관리합니다. 초기 스키마 생성, 컬럼 추가 등의 이력을 남겨 개발자 협업 시 혼선을 줄입니다.

4. 향후 확장 고려사항

- **Elasticsearch 도입:** DB 전환 후에도 검색 성능 향상이 필요하다면, Elasticsearch/OpenSearch를 도입해 기사 본문이나 키워드 검색을 고속화할 수 있습니다 [40]. 이때 DB는 진실된 원본, ES는 색인 용도로 쓰는 식입니다.
- **AI 모델 데이터 저장:** LLM을 통한 **기사 점수 산정이나 추천 결과**를 저장/분석하려면 별도 테이블을 둘 수도 있습니다. 예컨대 기사별 추천 점수 로그, 키워드 추천 결과 등을 저장하여 품질 개선에 활용합니다 [42].
- **독자 피드백 루프:** 만약 독자용 사이트와 연결되어, 독자의 조회나 클릭 데이터를 수집하게 된다면, 그 데이터도 DB 테이블로 설계해야 합니다 [42]. 이는 QualiJournal의 알고리즘을 고도화하는 기반이 될 것입니다.

결론적으로, **DB 전환**은 시스템을 한층 발전시키기 위한 필수 단계이며, 면밀한 설계와 준비가 필요합니다. 다행히 현재 JSON 데이터 구조가 정형화되어 있어, 이를 SQL 스키마로 옮기거나 NoSQL에 투영하는 일이 비교적 수월할 것으로 예상됩니다 [3]. 개발자는 점진적인 마이그레이션 전략으로 위험을 낮추고, 운영자는 전환 단계에서 구 버전과 신 버전 데이터의 정합성에 특별히 주의해야 합니다. 이 가이드북의 미래 개정판에서는 실제 DB 전환 후의 구조와 운영 방법이 추가될 것입니다.

정기 운영 루틴 및 모범 사례

QualiJournal 시스템을 안정적으로 운영하기 위해서는 **일일/주기적인 점검과 작업 루틴**이 필요합니다. 아래는 운영자(편집자)와 개발자 관점에서 수행해야 할 **반복 실행 루틴**과 **베스트 프랙티스**를 정리한 섹션입니다. 이를 따르면 시스템 장애를 미연에 방지하고, 매일의 뉴스 발행을 원활하게 이어갈 수 있습니다.

1. 매일 아침 점검 루틴 (Morning Checklist)

목표: 전일 밤/당일 새벽에 자동으로 실행된 수집/발행 작업 결과를 확인하고, 금일 발행 프로세스를 시작할 준비를 합니다.

- **Cloud Scheduler 확인:** (자동화된 경우) 정해진 시간에 `/api/report`가 정상 호출되었는지 확인합니다. GCP Cloud Scheduler 대시보드에서 마지막 실행이 성공적으로 표시됐는지, 또는 Cloud Run 로그에서 해당 시간에 수집 작업 로그가 있는지 확인합니다.
- **로그 에러 스캔:** Cloud Run 로그에서 ERROR 또는 WARNING 레벨 로그가 발생했는지 살펴봅니다. 특히 새벽 자동 발행 시도에서 오류가 있었다면, 원인을 파악해야 합니다. (예: 외부 API 키 만료, 수집 소스 404 오류 등)
- GCP 로그 탐색기에서 `resource.type="cloud_run_revision" severity>=ERROR` 등의 쿼리로 최근 12시간 로그를 필터링하면 유용합니다.
- **백업 파일 존재 확인:** GCS 아카이브 버킷에 어제 날짜의 MD/HTML/JSON 파일이 저장되었는지 체크합니다. 만약 존재하지 않으면, 자동 발행이 제대로 되지 않은 것이므로 그날은 수동 발행을 진행해야 할 수 있습니다.
- GCS 콘솔에서 버킷을 열거나, `gsutil ls gs://qualijournal-archive/2025-10-13*` 명령으로 확인합니다.
- **키워드 결정:** 오늘 발행할 키워드를 정했는지 확인합니다.
- 만약 키워드 추천 알고리즘이 없다면 편집팀에서 전날 회의로 결정했을 것이고, 없다면 아침 회의에서 결정해야 합니다.
- 키워드가 정해졌으면 기록해두고, 시스템에 입력할 준비를 합니다. (UI에서 키워드 입력)
- **소스 업데이트:** 크롤링 소스(뉴스 RSS 등)에 변경이 없는지 주기적으로 살펴봅니다. 아침엔 아니더라도, 매주 한 번 정도 official_sources.json의 링크들이 유효한지 검사하거나, 새로운 소스를 추가할 필요가 없는지 검토합니다 [43].
- **시스템 헬스 체크:** Cloud Run 서비스 상태를 확인합니다 (콘솔에서 **Running** 상태인지, 최근 오류 리비전은 없는지). 메모리/CPU 사용이 전일 대비 급증하지 않았는지도 모니터링합니다. 필요 시 알림 설정을 조정합니다.

이러한 아침 점검을 통해 **전일의 문제를 조기에 발견**하고, 금일 발행에 차질이 없도록 준비합니다. 많은 항목이 자동화되어 있더라도, 사람의 점검이 중요한 부분을 캐치해낼 수 있습니다.

2. 일일 뉴스 발행 루틴 (Daily Publishing Routine)

목표: 정해진 시간까지 금일 키워드 뉴스 발행을 완료합니다. 예를 들어 **오전 10시 발행**이 목표라면, 이를 역산하여 일정을 운영합니다.

- **[T-알림] 키워드 입력 및 수집 시작:** 발행 2~3시간 전 (예: 오전 7~8시 경) 오늘의 키워드를 **관리자 UI에 입력**하고 "보고서 생성"을 실행합니다. (자동화되어 새벽에 수집되었다면 이 단계 생략)
- `/api/report` 호출이 성공하면 UI에 기사 목록이 뜹니다. 만약 해당 시점 수집 결과가 부족하면, 수동으로 다시 collect를 시도하거나 키워드를 세분화/변경 고려합니다.
- **[T-2h] 기사 검토 및 1차 선정:** 편집자는 수집된 기사들을 **훑어보고 가치 있는 기사들에 우선 승인** 표시를 합니다. UI에서 점수와 요약을 참고하여 15~20개 정도를 골라 ✔ **승인**합니다.
- 너무 중복된 내용이나 품질 낮은 글은 ✖ **제외**하고 리스트에서 감춥니다. 애매한 것은 보류 (아무 조치 안 한 상태)로 둡니다.

- 이 과정에서 이미 요약이 필요한 영문 기사들은 내용을 파악하기 어려우므로, 우선 한국어 기사들 위주로 검토합니다.
- **[T-1.5h] 요약/번역 실행:** 아직 요약/번역이 안 된 기사들에 대해 **일괄 요약** API를 실행합니다 (`/api/enrich/keyword`). UI에서 "요약 생성" 버튼 클릭 → 스피너 동작 → 완료되면 모든 기사 카드에 한국어 요약이 채워집니다.
- 편집자는 요약 결과를 읽어보며, 잘못되거나 어색한 부분을 **편집자 코멘트** 필드에서 수정합니다. 예컨대 번역투 문장을 다듬거나, 중요한 정보를 추가합니다.
- 일부 기사 요약이 마음에 들지 않으면, 그 기사만 선택해 `/api/enrich/selection`으로 재생성하거나 수동으로 편집합니다.
- **[T-1h] 최종 선정 및 정렬:** 요약을 모두 확인한 후, **최종 발행할 15개 내외의 기사를 결정**합니다. 필요하면 보류 중인 기사도 추가 승인해서 15개 이상을 확보합니다.
- UI에서 기사 정렬 순서를 조정할 수 있다면, 독자들이 읽기 좋은 순서 (예: 중요도 순 또는 카테고리별)로 재배치합니다. 현재 UI는 기본 점수순이지만, 섹션별 묶음 등 수동 조정이 필요할 수 있습니다. (UI가 지원하지 않으면 Markdown export 후 수동 편집으로라도 순서 조정 가능)
- 모든 기사에 편집자 한줄 의견이 입력되었는지 확인합니다 (빈 코멘트 있으면 기본 요약이라도 넣는 것이 좋습니다).
- **[T-0.5h] 미리보기 및 발행 준비:** "미리보기" 버튼을 눌러 **최종 페이지를 확인**합니다. 모달에 표시된 Markdown 렌더링을 보며 레이아웃이나 내용에 이상은 없는지 살핍니다. (예: 제목 오탈자, 줄바꿈 문제, 이미지 깨짐 등이 없는지)
- 문제가 있다면 UI 편집 화면으로 돌아가 수정하고, 다시 미리보기를 확인합니다.
- 모두 만족스럽다면 **발행(Export)**을 실행합니다. Markdown을 다운로드 받아 내부 기록으로 저장하고, 필요시 HTML도 다운로드합니다. (또는 publish API가 있다면 호출)
- **[T=0h] 발행 공지:** 정해진 발행 시각에 최종 산출물을 독자들이 볼 수 있는 곳에 올립니다. QualiJournal이 별도 독자 사이트로 자동 게시된다면 이걸로 끝이지만, 만약 수동 게시라면:
- 새 Markdown을 사내 위키나 블로그 플랫폼에 업로드하거나,
- 사내 이메일/메신저로 해당 파일을 전송하여 공유합니다.
- SNS 등에 링크를 배포할 경우, 게시물 작성 및 공유도 진행합니다.
- **사후 기록:** 발행 완료 후, 해당 날짜 키워드와 주요 이슈를 운영 노트에 남겨둡니다. (예: 2025-10-14 "인공지능" - 기사 18개 수집, 15개 발행, 주요 이슈: OpenAI GPT-4 발표 등.) 이는 나중에 **아카이브 검색**이나 **트렌드 분석**에 참고됩니다.

3. 주간/월간 유지보수 루틴

- **주간 로그 리뷰:** 매주 한 번 전체 로그를 훑어보기. 빈번히 발생하는 경고는 없는지, 외부 API 호출 추이는 어떤지 점검합니다. 예를 들어 OpenAI API 사용량이 증가하면 키 할당량 증설을 고려할 수 있습니다.
- **시스템 업데이트:** 격주 혹은 월별로 라이브러리 버전을 최신으로 올리고, Docker base image 최신화, OS 패키지 업데이트 등을 시행합니다. CI 파이프라인에서 새 브랜치로 실험해보고 main에 반영합니다.
- **백업 검증:** 아카이브에 쌓인 파일을 몇 개 골라 열어봅니다 (Markdown 렌더링, JSON 파싱)해서 정상인지 확인합니다. 특히 오래된 파일이 손상되진 않았는지, GCS lifecycle이 의도대로 작동하는지 확인합니다.
- **훈련 및 문서:** 편집자 교대나 신규 투입 시를 대비하여, **운영 매뉴얼**(바로 이 문서)을 최신 상태로 유지하고 필요한 부분을 교육합니다. 또한 개선 요청이나 발견된 버그를 수집해 개발 백로그에 전달합니다.

4. 예외 상황 대처

- **컨텐츠 부족 비상:** 특정 키워드로 15개 미만 기사만 수집되어 발행 기준 미달인 경우 [14], 우선 자동 보충 (`augment_from_official_pool`)이 되었는지 확인하고 그래도 부족하면:
- 다른 키워드로 변경을 긴급히 검토하거나 (주제가 너무 마이너했을 수 있음),
- `require_editor_approval` 옵션을 꺼서 15개 미만이라도 발행을 강행합니다 [17]. 단 이때 발행 품질이 낮아질 수 있으므로, 사전에 공지하거나 사후 보완 조치를 생각합니다.
- **시스템 장애 시:** Cloud Run 서비스 오류로 UI/API 사용이 불가능하면:

- 즉시 GCP Status를 확인하고, 문제가 길어질 경우 **로컬 백업 방법**으로 전환합니다. (예: local PC에 저장된 repository에서 `run_quali_today.ps1` 같은 스크립트를 수동 실행 [44] 하여 결과 JSON/MD를 뽑아내는 비상시나리오)
- UI 없이 JSON 편집으로 돌아가야 할 수도 있으므로, 편집자들에게 JSON 편집 방법을 사전에 교육하거나, 최소한 CSV export된 데이터를 Excel로 보고 판단하는 법을 공유합니다.
- 장애 복구되면 root cause를 분석하고, CI/CD나 테스트에서 잡아내지 못한 원인을 문서화합니다.
- **API 한도 초과:** 요약/번역 API 호출 횟수가 초과되어 실패할 경우, 무료 요약 기능만 사용하거나 (ex: GPT-3.5로 degrade) 해당 기사들을 수동 요약하는 식으로 대응합니다. 그리고 즉시 키 상향이나 요금 플랜 조정을 고려합니다.

5. 협업 및 커뮤니케이션

QualiJournal 운영은 기술팀과 편집팀의 협업이 요구됩니다: - 편집팀은 매일 키워드 선정 및 기사 품질 관리의 역할을 맡고, 기술팀(개발자)는 시스템이 원활히 작동하도록 백그라운드에서 지원합니다. - 서로 슬랙 채널이나 메신저로 **매일 발행 현황을 공유**하면 좋습니다. 예: "오늘 키워드: 인공지능, 18개 수집/15개 발행, 특이사항 없음" 등의 메시지. - 새로운 기능이 추가되거나 UI/프로세스가 바뀌면, 개발자는 간단한 **공지/가이드 업데이트**를 해서 편집자가 혼란 없이 받아들일 수 있게 합니다. - 주기적인 회고 미팅을 통해, 한 주의 발행 경험에서 나온 개선 아이디어 (예: UI 버튼 위치 변경, 특정 소스 추가 요청 등)을 수렴하고 우선순위를 정해 개발 로드맵에 반영합니다.

위 운영 루틴을 습관화하면, QualiJournal 시스템은 **안정적인 일일 발행 사이클**을 유지할 수 있습니다. 처음에는 다소 체크리스트가 많아 보여도, 익숙해지면 대부분 자동화되거나 빠르게 처리 가능한 일들입니다. 중요한 것은 **일관성(consistency)**이며, 이는 독자들에게 매일 고품질의 뉴스를 제공하는 기반이 될 것입니다.

이상으로 QualiJournal 관리자 시스템에 대한 **최종 가이드북**을 마칩니다. 이 문서의 내용대로 따라하면 개발자는 시스템을 구축/배포/개선할 수 있고, 운영자(편집자)는 매일의 뉴스를 만들어낼 수 있습니다. QualiJournal를 통해 한 층 더 효율적인 **지식 큐레이션과 전달**이 이루어지길 기대합니다. 필요한 경우, 향후 업데이트에 따라 본 가이드북도 개정하며, 질의사항은 담당 개발팀에 문의하시기 바랍니다. 감사합니다. [1] [45]

---

[1] [4] [5] [6] [7] [10] [13] [14] [15] [16] [17] [18] [23] [26] [38] [39] [42] [44] [45] 1004_2A고도화report.pdf
file://file-Ko3WmavWUUxUmLMC1s6fso

[2] [3] [8] [9] [11] [12] [19] [20] [21] [35] [36] [40] 1003_5A퀄리뉴스를 전면적으로 리팩터링하여 안정적이고 확장 가능한 시스템으로 구축하려면.pdf
file://file-C1X8pMSvAVd6Xs392FEnwq

[22] [24] [25] [27] [28] [29] [30] [31] [32] [33] [34] [41] 1003_4UI.pdf
file://file-Gcxm7FdLSYTHtsLZnDwtiJ

[37] [43] 1004_3A작업계획report.pdf
file://file-S3G4TQFPei3GeAJcXjN6B3