ChatGPT

Implementation Guide: QualiJournal Admin Report, Summarization, and Export Features

1. Overview of Required Functions

The QualiJournal Admin system needs three major features implemented to complete the editorial workflow: **daily report generation**, **news card summarization & translation**, and **result export** [1]. These functions were identified as missing in the mid-term plan and are crucial to allow editors to produce a daily keyword news report, automatically summarize/translate collected news cards, and export the final curated list for backup or analysis [1]. In summary, the tasks are:

- **Daily Report Generation:** Compile all editor-approved articles into a Markdown report (one report per day/keyword) for archival and sharing.
- **News Card Summarization/Translation:** Automatically generate concise summaries (and translate them if needed) for news articles either for all collected items of a keyword or for just the final selected items.
- **Export of Final Selections:** Provide one-click download of the final list of selected articles in **Markdown** and **CSV** formats for external use.

These features will fill the functional gaps and make the admin workflow self-contained – from collecting and approving news to publishing reports and exporting data [1]. All implementations must be **idempotent** – running them multiple times should not create duplicate entries or corrupt data. Instead, each run should either overwrite previous outputs or detect existing processing to avoid duplication. This ensures consistency even if an action (e.g. generating a report or summary) is triggered twice.

2. Backend API Design

We will introduce new FastAPI endpoints for each feature. All endpoints will be **secured for admin use only**, meaning they require the valid admin token (as already configured in the system) to be accessed [2]. Each endpoint will trigger the corresponding backend script or logic and return a result (either a file path/URL or a file download). Below is a summary of each API and its behavior:

`POST /api/report` – Generate Daily Report

This endpoint triggers the creation of a daily report in Markdown format. When called, the server will internally run the `tools/make_daily_report.py` script, which gathers all articles that the editor has approved in the current cycle (from `selected_articles.json`) and composes them into a Markdown report [3]. The report typically includes the date, the keyword of the day (if applicable), and a list of the selected news items with their titles, summaries, and any editor notes.

Output: The script will save the Markdown report file under the reports archive directory, e.g. `archive/reports/{YYYY-MM-DD}_{keyword}.md` (date and keyword in filename) [4]. The endpoint responds with a JSON containing the path or URL to the generated report file [5]. For example:

```
{ "report_path": "archive/reports/2025-10-07_5G.md" }
```

This allows the frontend to present a link for downloading or viewing the report. (The previous GET variant of `/api/report` used for testing will be deprecated in favor of this POST trigger [5].)

Implementation details: In FastAPI, implement this as a `@app.post("/api/report")` handler. Inside, load or determine the current keyword and date (for naming). For idempotency, ensure the script overwrites any existing report for the same date/keyword rather than appending, so running it twice in one day won't duplicate content. You can invoke the report script using Python's subprocess or by calling a function:

```
import subprocess
result = subprocess.run(["python", "tools/make_daily_report.py"], capture_output=True)
if result.returncode != 0:
    logger.error("Report generation failed: %s", result.stderr.decode())
    raise HTTPException(status_code=500, detail="Report generation failed.")
# Assume script outputs or knows the file path
report_path = determine_report_path()  # e.g. use date/keyword or parse stdout
return {"report_path": report_path}
```

Make sure to create the output directory (`archive/reports`) if it doesn't exist and open the file in write mode (the script should handle this) to avoid accumulating duplicate entries. The script likely reads `selected_articles.json` (which contains entries marked `approved=true`) and writes a well-formatted Markdown file. The Markdown should include a header (title with date/keyword), followed by the list of approved articles grouped by category or source. Each article entry can list the title (possibly hyperlinked), a brief summary (or the article's summary text, possibly translated), and any editor note. This way, the daily report is a self-contained summary of the day's curated news.

`POST /api/enrich/keyword` – Summarize & Translate All Collected Articles

This endpoint performs **bulk summarization/translation on all collected news articles for the current keyword** (regardless of whether they were approved by the editor) [6]. It calls `tools/enrich_cards.py` with a mode to process the entire set of articles gathered for the keyword. The script will iterate through each article's title and content, extract key sentences, and generate a short summary. If the source content is in English, it should translate the summary into Korean (using an API like OpenAI or a translation service, if API keys are available) [7] [6]. This provides the editor with a quick overview of all content without reading each in full.

Output: The summarization results for all articles are saved to a file, for example under an `archive/enriched` directory. A suggested naming is `archive/enriched/{keyword}_{YYYY-MM-DD}_all.json` or `.md` [8]. The file would contain each article's ID or title and its generated summary/translation. The endpoint returns a JSON with the path to this summary file [6]. (Using JSON format for the output can be convenient for programmatic use, while Markdown/text is convenient for human reading. You may choose either; for a quick readable result, a Markdown file listing each article and its summary is useful.)

Implementation details: Implement as `@app.post("/api/enrich/keyword")`. Internally, invoke `enrich_cards.py` with the appropriate context (perhaps passing a `--mode=all` or similar). Ensure

the script knows which keyword to process – this could be derived from a global state or a file (for instance, if `selected_keyword_articles.json` holds the current keyword and list of all collected articles). The script should summarize each article only once; if run multiple times, it should overwrite or update previous summaries rather than duplicate them. For idempotency, you might have the script check if an article already has a summary and skip or overwrite it. The output file itself should be overwritten on each run (open in write mode) so the content stays unique. If the summarization is time-consuming, consider setting a reasonable timeout on the subprocess (or using FastAPI's background tasks). For now, a synchronous call is acceptable, but be mindful of not letting it run indefinitely (you can use Python's `subprocess.run(..., timeout=seconds)` to prevent hanging). If an error occurs (e.g., an external API failure), catch it and return an HTTP 500 with an error message; also log the exception for debugging.

`POST /api/enrich/selection` – Summarize & Translate Selected Articles

This endpoint is similar to the above but operates **only on the editor's final selected articles** [6]. It calls `enrich_cards.py` in a mode to process just the approved list (for example, those in `selected_articles.json`). The purpose is to generate summaries/translations for the curated subset, which might be useful for directly including in the final report or for quick review of chosen articles in the target language.

Output: The results are saved to a file similar to the above, e.g. `archive/enriched/{keyword}_{YYYY-MM-DD}_selected.json` (or `.md`). The endpoint returns a JSON with the file path [9]. The content format is the same (each selected article with its summary/translation).

Implementation details: Implement as `@app.post("/api/enrich/selection")`. The logic is analogous to `/api/enrich/keyword`, but filtering the input to only approved items. Likely the script can read `selected_articles.json` to get the list of approved article IDs/titles and then summarize those. Ensure overwriting of the output file on each run for idempotency. If a summary file from a previous run exists, it should be replaced. (In practice, you might combine the implementations of the two enrich endpoints by passing a parameter to the script indicating which set to process, but exposing them as two endpoints gives flexibility in the UI.) As with the keyword version, handle errors and long execution appropriately (e.g. spinner on the UI, and possibly a timeout on the server side for the subprocess call).

Note: Both enrich endpoints are synchronous in this implementation. In the future, these could be offloaded to background tasks for asynchronous processing, especially if using heavy AI APIs. For now, we will keep it simple: the request will hang until completion, and the UI will show a loading indicator (see SSE/Spinner plan below) [10].

`GET /api/export/md` – Export Final Selection as Markdown

This endpoint allows the editor to download the **final list of published articles in Markdown format** [11]. The "final selection" refers to the set of articles that have been approved (and presumably published) in the system. The endpoint will compile those entries into a Markdown text, formatted as a simple list or document.

Output: A Markdown file is returned in the HTTP response. The content should include each article's key details in a readable format – for example: title, a short summary or description, the source or link, and any editor comment (if provided) [11]. Essentially, this is a dump of the final selection in markdown table or list form. One could format it as a bulleted list of articles:

```
- **Article Title 1** – *Source Name*, 2025-10-07
  Summary: ...
  Editor's Note: ...

- **Article Title 2** – *Source Name*, 2025-10-07
  Summary: ...
  Editor's Note: ...
```

Ensure that all relevant fields (title, link, summary, note) are present in the Markdown template [11]. The response should set appropriate headers so that the browser will download it as a `.md` file (e.g. `Content-Type: text/markdown` and a `Content-Disposition: attachment; filename="selection.md"`).

Implementation details: Implement as `@app.get("/api/export/md")`. When this endpoint is hit, the server should read the current final selection data. This could be from `selected_articles.json` (and possibly the community selection file if those are separate). Merge the data if needed to get one unified list of all final approved articles (the JSON likely already contains both official and community articles marked approved). Then generate a Markdown string. Use the same ordering or grouping of items as the admin UI if applicable (you might group by category or source, or just list all). For safety, ensure any special characters are properly escaped in Markdown. Finally, return the text content. For example:

```
from fastapi import Response
md_content = generate_markdown_from_selection()  # your function to build the MD text
return Response(md_content, media_type="text/markdown",
                headers={"Content-Disposition": "attachment; filename=selection.md"})
```

This will prompt the user to download a file named `selection.md`. Include a BOM at the start only if needed for Markdown (usually not necessary for UTF-8 text files; BOM is more crucial for Excel CSV, see below). Since Markdown is UTF-8 text, it should display fine in editors without a BOM, but adding `\ufeff` at the beginning is harmless if you want to be consistent.

`GET /api/export/csv` – Export Final Selection as CSV (with BOM)

This endpoint provides the final selection data as a CSV file [11]. Each row in the CSV will correspond to one article in the final selection. The columns can be: Title, Source (or Domain), Publication Date, URL, Summary, Editor Note – or a subset of these that are most useful. Decide on the schema that best fits the data (for example, you might exclude the full summary to keep the CSV concise, or include it if needed for analysis).

Output: A CSV file download. The response must include a **UTF-8 BOM (Byte Order Mark)** at the beginning of the file [12]. The BOM (`\ufeff`) ensures that if the CSV is opened in Excel, it will correctly interpret UTF-8 encoding (preventing garbled text for non-ASCII characters) [12]. So the content might look like:

```
\ufeffTitle,Source,PublishedDate,URL,EditorNote
"Foxconn Chief Procurement Officer Honored...", "iConnect007", "2025-10-07",
```

```
"https://iconnect007.com/article/...", ""
"Taiwan PCB Industry Projects NT$915.7B Output...", "iConnect007",
"2025-10-07", "...", ""
...
```

All fields that could contain commas or special characters should be quoted. Use a CSV utility or Python's `csv` module to generate the content to ensure proper escaping.

Implementation details: Implement as `@app.get("/api/export/csv")`. Similar to the Markdown export, read the final selection data and generate CSV lines. You can use an in-memory approach:

```
import io, csv
output = io.StringIO()
writer = csv.writer(output)
writer.writerow(["Title", "Source", "Date", "URL", "EditorNote"])  # header
for article in selected_articles:
    writer.writerow([article["title"], article["source"], article["published_date"], article["url"],
article["editor_note"]])
csv_data = output.getvalue()
output.close()
# Prepend BOM
csv_data = '\ufeff' + csv_data
return Response(csv_data, media_type="text/csv",
                headers={"Content-Disposition": "attachment; filename=selection.csv"})
```

This will prompt a download of `selection.csv`. By adding the BOM (`\ufeff`) at the start of the content, Excel will recognize the file as UTF-8 [12]. (If the system already had an implementation, ensure to fix it by adding the BOM as noted in the requirements.)

Note on idempotency: The export endpoints are stateless reads – calling them repeatedly simply regenerates the same output from the current data, so there's no risk of duplication. Just ensure the order or formatting remains consistent across calls.

3. File Output Paths and Formats

Understanding where each output is stored and in what format is important for both implementation and deployment:

- **Daily report Markdown** – Stored in the path defined by config for reports. By default, this is `archive/reports`. The file name should include the date (and keyword if applicable) to avoid collisions [4]. For example: `archive/reports/2025-10-07_반도체.md` for a report on keyword "반도체" on 2025-10-07. The Markdown format should have a clear title (e.g., "# Daily QualiNews Report: 2025-10-07 (Keyword: 반도체)") and then sections or bullet points for each article. Grouping by category (policy, general news, AI news, community, etc.) is recommended to mirror the UI categorization [13] [14], but a simple list is acceptable if grouping is not readily available in the data. Each entry might look like a bullet with **title** (as a link), followed by a short summary or excerpt, and an editor comment if present. Ensure that if an article appears multiple

times (e.g., duplicates from different sources), it's only listed once in the report – filtering out duplicates by title or URL can be done in the script as a precaution.

- **Summarization output** – Saved in `archive/enriched` (you may need to create this directory and possibly add it to your config paths). There can be two kinds of files: one for the entire keyword (`..._all`) and one for the selected subset (`..._selected`). Using JSON format is useful for structured data (each article ID mapping to summary text), whereas Markdown/text is more human-readable. The handover suggests either JSON or MD; for immediate use, a Markdown file containing all summaries in a list form is convenient [8]. For example, `archive/enriched/반도체_2025-10-07_all.md` might contain:

```
# Summaries for keyword "반도체" (All collected articles)
- **Foxconn Chief Procurement Officer Honored...** – 요약: 폭스콘 최고 조달 책임자가 2025년 공급망 리더 Top 30에 선정되었다... (translated summary)
- **Taiwan PCB Industry Projects NT$915.7B...** – 요약: 대만 PCB 산업이 2025년에 NT$9157억 규모를 예상하며 AI 수요가 성장을 견인...
...
```

and `archive/enriched/반도체_2025-10-07_selected.md` would similarly list only the approved articles. If JSON is used instead, it could look like:

```
{
 "keyword": "반도체",
 "date": "2025-10-07",
 "summaries": [
  { "id": "<article-id-1>", "title": "Foxconn Chief Procurement Officer...", "summary_ko": "폭스콘 최고 조달 책임자가 ...", "summary_en": "Foxconn's CPO was honored ..."},
  ...
 ]
}
```

Choose the format that best suits how the editors will consume this data (likely they will open and read it, so Markdown or text is user-friendly).

Ensure the filenames incorporate the keyword and date so that multiple runs for different days/keywords produce distinct files. If the same keyword is processed on the same day more than once, your script can either overwrite the file (keeping only the latest summaries, which is usually fine and maintains idempotency) or version the filename with a suffix like `_v2` – however, simplest is to overwrite since the input data presumably hasn't changed within the same day. Always write fresh content to avoid appending duplicate summary lines.

- **Export files (Markdown & CSV)** – These are not stored on the server permanently by our design; they are generated on-the-fly and returned to the client. The Markdown export content mirrors much of the daily report format (title, summary, link, note), but it might not be exactly the same as the daily report if the daily report is keyword-centric. The export is meant to capture all final selections to date or for the current batch. Clarify whether "current final selection" means just today's selection or a cumulative archive. Given the context, it sounds like it's the current set of selected articles that are about to be (or have been) published for the current keyword/day.

Therefore, the export MD/CSV will be similar in scope to the daily report, but provided in different formats for external use [11]. No specific path is needed since we deliver it directly, but if needed, you could save a copy in `archive/exports/` for logging or re-download. This is optional – the primary requirement is to deliver the file to the user on request.

For CSV, as noted, always prepend the UTF-8 BOM and use proper quoting. The BOM is critical for Excel; it's explicitly required because without it, opening the CSV in Excel might mangle non-English text [12]. We include it in the response content (FastAPI's Response as shown earlier). Test the CSV by opening it in Excel to confirm that Korean or other Unicode characters display correctly, and adjust if necessary.

4. FastAPI Integration and Script Invocation

Integrating these features into the FastAPI backend involves adding the new routes and ensuring the scripts are executed correctly. Here are the key steps and considerations:

- **Define the new routes:** In your server code (e.g., `server_quali.py` or wherever the FastAPI app is defined), add the function decorators for each endpoint:

```
from fastapi import FastAPI, HTTPException
app = FastAPI()

@app.post("/api/report")
def create_report():
    ...

@app.post("/api/enrich/keyword")
def enrich_all():
    ...

@app.post("/api/enrich/selection")
def enrich_selected():
    ...

@app.get("/api/export/md")
def export_md():
    ...

@app.get("/api/export/csv")
def export_csv():
    ...
```

Ensure these are added under the appropriate router or directly to `app` if not using routers. All should be protected by whatever authentication/authorization the app uses (e.g., checking an `Authorization` header for a token). If there is a dependency or middleware for admin token, make sure these endpoints are covered. For example, if using a simple token check, do something like:

```
from fastapi import Depends, Header
ADMIN_TOKEN = os.getenv("ADMIN_TOKEN")
```

```
def verify_admin(token: str = Header(None)):
    if token != ADMIN_TOKEN:
        raise HTTPException(status_code=401, detail="Unauthorized")

@app.post("/api/report", dependencies=[Depends(verify_admin)])
def create_report():
    ...
```

This ensures only authorized calls proceed [2].

- **Invoke external scripts or functions:** The core logic for each endpoint will likely delegate to existing scripts (`make_daily_report.py` and `enrich_cards.py`). You have a few choices for invocation:

- **Subprocess call:** This is straightforward – run the script in a new process. As shown earlier, `subprocess.run(["python", "tools/make_daily_report.py"])` will execute the script. This keeps the script's execution isolated. However, capture the output and return values. If the script prints the output file path or any status messages, capture them via `capture_output=True` so you can use them. Also handle `returncode` to detect errors. Example for report:

```
proc = subprocess.run(["python", "tools/make_daily_report.py"], capture_output=True, text=True)
if proc.returncode != 0:
    raise HTTPException(status_code=500, detail=f"Report script failed: {proc.stderr}")
# If the script prints the report path to stdout, we can parse it:
report_path = proc.stdout.strip()
return {"report_path": report_path}
```

If the script does not print the path, you may have to construct it (e.g., using current date and keyword as shown in section 3).

- **Import and call directly:** If the scripts are written with a `main()` function or similar, you can import the python module and call a function to generate the content. For example:

```
from tools import make_daily_report
report_path = make_daily_report.generate_report()  # hypothetical function
```

This approach can be faster (no new process overhead) and easier to get return values directly. However, it may require refactoring the script to expose functions. Also, ensure that any relative file paths in the script are correct (the working directory might differ when called within FastAPI). You might need to adjust `os.chdir()` to the project root or use absolute paths.

Given the developer's familiarity and for simplicity, the **subprocess approach** is likely sufficient for now. It mirrors how a user might run the script manually, and by keeping it separate, we avoid blocking the FastAPI event loop beyond the subprocess call (which FastAPI will handle in a thread). **Important:** if using subprocess, include proper error handling and logging: - Wrap calls in try/except for

`subprocess.CalledProcessError` if using `check=True`. - Log standard output and errors for debugging. - Possibly set a timeout (e.g., `subprocess.run(..., timeout=300)`) especially for the enrich tasks which might call external APIs and take time [15]. - If a timeout happens or the script throws, catch that and return a 500 with an informative message to the UI.

- **Use config.json for paths:** To avoid hard-coding paths in your new code, read from `config.json` (which is likely loaded at startup and merged into some global config object or accessible via orchestrator) [16]. For example, `config["paths"]["reports"]` gives the reports directory, `config["paths"]["selection_file"]` gives the path to `selected_articles.json`. The scripts may already use these config values internally (if they import orchestrator or a common config loader). If not, you can safely assume default locations as given. Still, linking to config ensures that if the paths are changed in config.json, your endpoints automatically follow those changes. Likewise, if there are settings like output formats or thresholds in config, respect them (e.g., `features.output_formats` might include "md" and "csv", confirming these features are enabled [17]).

- **Prevent duplicate processing:** Idempotency should be built-in:

- For `/api/report`: If the day's report already exists and nothing changed, running again just regenerates the same file. Overwrite the file to ensure it doesn't append. Optionally, you could check if the file exists and skip generation to save time, but only do that if you are certain the content would be identical. It may be safer to always regenerate (since the editor might have added a new approved article since last run).
- For `/api/enrich/...`: The script could skip already summarized articles, but since we want the latest translations or maybe improved summarization, it's fine to regenerate each time. Just clear or overwrite previous summary output. If the enrich script writes back to each article's data (for example adding a `summary_ko` field in the JSON), ensure it updates those fields rather than duplicating them (no creating multiple summary entries per article).

- For `/api/export`: Since it's generated on demand, there's no persistence issue. If you decide to cache or save the exported file, be careful to update it when data changes. Otherwise, always computing fresh from `selected_articles.json` ensures no stale or duplicate data.

- **Logging and debugging:** Integrate these endpoints with your logging system. Log when a generation starts and ends, and any errors. For example:

```
logger.info("Generating daily report...")
...
logger.info("Report generated at %s", report_path)
```

Similarly for summaries and exports. This will help in troubleshooting by providing a timeline in logs.

- **Testing hooks:** If possible, code the core logic in a way that it can be invoked for testing without actual HTTP calls (for instance, factor out the part that calls the script and returns a path into a helper function). This will ease writing Pytest tests that simulate the process (see the Testing section below). For example, have a function `generate_report_file() -> str` that your endpoint calls; your test can call that directly or use the test client to hit the endpoint and then verify the output file exists.

5. Frontend Integration

With the backend in place, the next step is to add UI controls in `index.html` (the admin dashboard) for the editor to use these features. The developer is expected to insert new buttons and corresponding JavaScript logic to call the new endpoints and handle responses.

Button Insertion Points in `index.html`

Examine the structure of `index.html` to determine where the new buttons should go: - For the **Daily Report generation** (`/api/report`): A logical place is at the top of the dashboard or in a toolbar where other global actions reside. The PDF suggests adding a button labeled "일일 보고서 생성" (Daily Report Generation) on the dashboard screen [18]. If there is a section showing the current keyword or date, placing the button next to it would make sense. For example, if the top of the page has a header with the current keyword, you might insert:

```
<button id="btn-generate-report">일일 보고서 생성</button>
```

in that header or controls div.

- For the **Card Summarization/Translation** (`/api/enrich/...`): The suggestion is to add a "카드 요약/번역" button for each section of articles [19]. If the UI lists articles in different categories (like Official News vs Community, or if it's segmented by sections), you might put a button at the top of each list. For instance, above the list of collected articles (to summarize all) and maybe above the list of selected articles (to summarize just those). If the UI is not clearly segmented, another approach is to provide two buttons in a relevant place:
- "요약/번역 (전체)" to call `/api/enrich/keyword` for all collected news.
- "요약/번역 (선정된 기사)" to call `/api/enrich/selection` for only the final picks.

However, to keep things simple, you might expose just one button if needed. The handover specifically references both endpoints, so ideally provide a way to trigger both. This could be two separate buttons or a single button that perhaps does one or the other based on context. For clarity, two buttons (e.g., "Summarize All" and "Summarize Selected") can be used, or one button with a dropdown to choose the scope.

In code, e.g.:

```
<div class="section-header">
  <h3>All Collected Articles</h3>
  <button id="btn-enrich-all">카드 요약/번역 (전체)</button>
</div>
<!-- list of articles -->
...
<div class="section-header">
  <h3>Selected Articles</h3>
  <button id="btn-enrich-selected">카드 요약/번역 (선정)</button>
</div>
<!-- list of selected articles -->
```

Adapt to how the HTML is structured. The key is the button IDs and labels.

- For the **Export buttons** (`/api/export/md` and `/api/export/csv`): The plan is to add either two buttons "Export MD" and "Export CSV" or a single dropdown menu containing those options [20]. Since the functionality is straightforward, two small buttons can be simplest:

```
<button id="btn-export-md">Export MD</button>
<button id="btn-export-csv">Export CSV</button>
```

  These could be placed in a toolbar area of the dashboard, perhaps alongside the report button or in a header/footer. They might logically sit near a "Publish" or "Finalized" indicator if one exists, or simply at the top-right corner of the page for utility. The exact location can be adjusted for UI/UX, but ensure they are visible and not confusing. (If there are many buttons, grouping the export ones in a `<div class="export-actions">` or similar is a good idea.)

After adding these elements in the HTML, style them if needed (you can reuse existing CSS classes for buttons). For example, if other buttons use a class like `class="btn btn-primary"`, apply that for consistency.

`fetch()` Logic and Modal/Link Rendering

We need to attach JavaScript logic to each new button to perform the API calls and handle the results. You can add a `<script>` block in `index.html` or in an existing JS file that runs on page load. Below is a breakdown per feature:

- **Daily Report Button (`#btn-generate-report`):**
  On click, initiate a POST request to `/api/report`. Use the Fetch API:

```
const reportBtn = document.getElementById('btn-generate-report');
const reportLink = document.getElementById('report-
link'); // assume an anchor for the link, added to HTML
reportBtn.onclick = async function() {
  reportBtn.disabled = true;
  showSpinner(); // function to display a loading spinner or message
  try {
    const resp = await fetch('/api/report', {
      method: 'POST',
      headers: { 'Authorization': 'Bearer <ADMIN_TOKEN>' }
    });
    if (!resp.ok) throw new Error(`Status ${resp.status}`);
    const data = await resp.json();
    console.log('Report generated at:', data.report_path);
    // Display link to the report:
    reportLink.href = data.report_path;
    reportLink.textContent = "보고서 열기"; // "Open Report"
    reportLink.style.display = 'inline'; // make it visible
  } catch (err) {
    alert("보고서 생성에 실패했습니다. \nError: " + err.message);
  } finally {
```

```
    hideSpinner();
    reportBtn.disabled = false;
  }
};
```

In the above pseudo-code, we assume:

- There is an `<a id="report-link" target="_blank" style="display:none;"></a>` in the HTML that will be used to show the report link.
- `showSpinner()` and `hideSpinner()` are functions to toggle a spinner graphic or loading text. If not already present, you can implement a simple spinner by, for example, adding a CSS class to the button or showing a small loading indicator element.
- The `Authorization` header is included if your backend requires the admin token. Replace `<ADMIN_TOKEN>` with the actual token mechanism (maybe stored in a JS variable or cookie). If your setup injects a token via a meta tag or global JS variable, use that. If you're not using a token on local deployments, you can omit the header.

Once the response comes back with `data.report_path`, we use it to set an anchor href. We set `target="_blank"` to open in a new tab when clicked (so the Markdown can be viewed in the browser if the server serves it, or downloaded, depending on the browser's handling of `.md` files). Alternatively, instead of showing a link, you could programmatically open the report in a new tab:

```
window.open(data.report_path, '_blank');
```

This would immediately open the report file once ready. However, providing the link in the UI gives the user the choice to click when ready and also see the path (which might reassure them the file was created).

- **Summarize All Button (`#btn-enrich-all`):**
  This will call `/api/enrich/keyword`. The logic is similar to the above:

```
const enrichAllBtn = document.getElementById('btn-enrich-all');
const enrichAllLink = document.getElementById('enrich-all-link'); // an <a> to open the summary results
enrichAllBtn.onclick = async () => {
  enrichAllBtn.disabled = true;
  showSpinner();
  try {
    const resp = await fetch('/api/enrich/keyword', {
      method: 'POST',
      headers: { 'Authorization': 'Bearer <ADMIN_TOKEN>' }
    });
    if (!resp.ok) throw new Error(`Status ${resp.status}`);
    const data = await resp.json();
    console.log('Keyword summaries file:', data.path);
    enrichAllLink.href = data.path;
    enrichAllLink.textContent = "요약 결과 열기";
    enrichAllLink.style.display = 'inline';
  } catch (err) {
    alert("요약 생성에 실패했습니다. Error: " + err);
```

```
  } finally {
    hideSpinner();
    enrichAllBtn.disabled = false;
  }
};
```

And similarly for **Summarize Selected Button (`#btn-enrich-selected`)** calling `/api/enrich/selection`:

```
const enrichSelBtn = document.getElementById('btn-enrich-selected');
const enrichSelLink = document.getElementById('enrich-selected-link');
enrichSelBtn.onclick = async () => {
  enrichSelBtn.disabled = true;
  showSpinner();
  try {
    const resp = await fetch('/api/enrich/selection', { method: 'POST', headers: {
'Authorization': 'Bearer ...'} });
    if (!resp.ok) throw new Error(`Status ${resp.status}`);
    const data = await resp.json();
    console.log('Selected summaries file:', data.path);
    enrichSelLink.href = data.path;
    enrichSelLink.textContent = "선정 기사 요약 열기";
    enrichSelLink.style.display = 'inline';
  } catch (err) {
    alert("요약 생성에 실패했습니다: " + err);
  } finally {
    hideSpinner();
    enrichSelBtn.disabled = false;
  }
};
```

Again, ensure there are `<a>` elements in the HTML to display or open the results. You might reuse a single link element for both or have separate ones. Alternatively, instead of links, you could initiate a download or show the content in a modal. Given the results are likely text, a nice enhancement would be to pop up a modal dialog with the summary content. That requires an extra step of fetching the file content:

```
const text = await (await fetch(data.path)).text();
showModalWithContent(text);
```

But this is an enhancement. Initially, just providing a link is simpler.

Remember to handle the case where these tasks take some time. The spinner or loading indicator should remain visible until completion. The UI should remain responsive – by disabling the button we prevent duplicate clicks. Also consider if the user navigates away or closes the page while waiting; since we're not using SSE here, that's fine (the fetch will just get canceled).

- **Export MD Button (`#btn-export-md`):**
  The export endpoints are GET and meant to trigger downloads directly. We have two approaches:

- **Direct link click:** Since these are GET endpoints, you can simply do `<a href="/api/export/md" download>` in HTML. The `download` attribute on an anchor will suggest the browser to download the response as a file. The PDF suggests using an anchor approach for simplicity [21] [22]. For example:

```
<a id="export-md-link" href="/api/export/md" download="selection.md"
style="display:none;">Export MD</a>
<button id="btn-export-md">Export MD</button>
```

Then in JS:

```
document.getElementById('btn-export-md').onclick = () => {
 document.getElementById('export-md-link').click();
};
```

This will effectively act like a direct download button. However, if an authorization token is required, this won't include it. One workaround is to embed the token in the URL (like `/api/export/md?token=XYZ`), but that is not secure or ideal. Alternatively, you can forego the token check for these export endpoints if you deem it low-risk and want the convenience (since they only serve data that an admin can see on the page anyway). If you keep security, the anchor method won't work unless you implement a cookie or session-based auth.

- **Fetch and create blob:** This method uses JavaScript to fetch the content, then creates a downloadable blob object. For example:

```
const exportMdBtn = document.getElementById('btn-export-md');
exportMdBtn.onclick = async () => {
 exportMdBtn.disabled = true;
 try {
  const resp = await fetch('/api/export/md', { headers: { 'Authorization': 'Bearer ...'} });
  if (!resp.ok) throw new Error(`Status ${resp.status}`);
  const blob = await resp.blob();
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = 'selection.md';
  document.body.appendChild(a);
  a.click();
  document.body.removeChild(a);
  URL.revokeObjectURL(url);
 } catch (err) {
  alert("Export MD failed: " + err);
 } finally {
  exportMdBtn.disabled = false;
 }
};
```

This will programmatically trigger a file download with the content from the response. It ensures the auth token is included. The code dynamically creates a hidden anchor and clicks it. This pattern can be reused for the CSV export (just change the endpoint and filename).

Considering the developer's experience, the fetch-and-blob approach is more flexible and secure. The anchor with `download` is simpler but may require adjusting auth (e.g., if the app is deployed on the same domain and the token is a cookie, an anchor might work). Choose based on your auth setup.

- **Export CSV Button (`#btn-export-csv`):**
  Similar to the MD case. If using direct anchor:

  ```
  <a id="export-csv-link" href="/api/export/csv" download="selection.csv"
  style="display:none;"></a>
  <button id="btn-export-csv">Export CSV</button>
  ```

  and JS to trigger the click. If using fetch:

  ```
  document.getElementById('btn-export-csv').onclick = async () => {
   // similar to above, fetch('/api/export/csv'), get blob, download as "selection.csv"
  };
  ```

  The CSV response should already have the BOM in it (added server-side), but you can double-check by reading the first few bytes of the blob if needed. Typically, just downloading the blob will preserve the BOM.

After implementing the above, test the UI flows: - Clicking "Daily Report" should disable the button and show a spinner; after a brief delay, a link or new tab should appear with the `.md` report. The UI can optionally display a success message or simply rely on the link appearing as indication. - Clicking "Summarize" should similarly show loading and then a link to the summary. Because these summaries might be something the editor wants to quickly glance at, consider opening them automatically (maybe in a new tab). You might decide that when the result is ready, you immediately do `window.open(data.path)` instead of showing a link that the user must click. This would pop open the summary text file. This is a UX decision – the documentation suggests displaying a link and possibly letting the user open it [19], but either is fine. - Clicking "Export MD/CSV" should trigger file downloads with no additional UI feedback needed (except maybe a quick flash as the browser starts download). If you want, you can show a spinner here too until the download starts, but typically downloads are fast. If the dataset is large, showing a spinner or at least disabling the button until response comes is wise.

Ensure to add any needed translations for button text if the rest of the UI is in Korean. For instance, "Export MD" might be fine as-is if the audience understands it, or you might label it "Markdown 내보내기".

Modal or In-Page Display (Optional)

The prompt mentions possibly using a modal or new tab to show the report content [23]. For the scope of this implementation, new tabs or downloaded files are simpler. If you wanted to get fancy: - You could create a modal dialog in HTML (a hidden `<div>` that you fill with content and show as an overlay) to display the report or summary text directly in the admin page. This avoids the context switch of going to another tab or downloading a file. Given that Markdown and text files are easy to render (you could even parse Markdown to HTML if you include a library or just display it as preformatted text), this could be a

nice-to-have. However, it's not strictly required. The guide suggests it as a consideration for user convenience [23] . - If time permits, implement a simple modal: after receiving the report content via fetch (instead of just the path), inject it into a modal `<pre>` or `<div>` and show it. Provide a close button. This way the editor can read the report immediately and decide if it's good, without leaving the admin interface.

For now, ensure the basic link/download mechanism works and is reliable.

6. SSE or Spinner Feedback Plan

Feedback to the user during long-running operations is critical. The system already uses **Server-Sent Events (SSE)** and JavaScript for some dynamic updates (like the KPI auto-refresh, possibly) – however, for these new features, the initial implementation will use a simpler approach with spinners or loading indicators [10] .

- **Current Approach (Spinner):** When a button is clicked (report generation or summarization), immediately indicate progress. This can be done by:
- Disabling the button to prevent duplicate clicks.
- Showing a spinner icon or a text like "Processing..." next to the button or in a common status area.
- Changing the cursor to a loading state (optional).

The spinner can be an existing CSS animation or a small GIF/inline SVG. If the project already has a global loading indicator (check if there is a `<div id="spinner">` or similar), use that. Otherwise, a quick solution is to use the button's `disabled` state styling (maybe greyed out) combined with an adjacent text. For example:

```
#loading-indicator { display: none; }
.loading { display: inline-block; }
```

and in JS:

```
document.getElementById('loading-indicator').classList.add('loading');
// do fetch...
document.getElementById('loading-indicator').classList.remove('loading');
```

This can be encapsulated in the `showSpinner()` / `hideSpinner()` functions mentioned earlier.

The idea is to inform the user that the system is working, since creating a report or summarizing can take several seconds [10] . On completion, hide the spinner and show the result (link or automatic download).

- **SSE Consideration:** In the future, if these tasks were executed asynchronously (in the background on the server), we could use SSE or WebSockets to push real-time progress updates to the frontend. For example, the server could stream events like "10% done... 50% done... done!" which the UI could use to update a progress bar. The current plan does not include implementing a full async queue, but the codebase mentions a potential `/api/tasks/...` system for background tasks [24] . If such a system were in place, the frontend could fire a request to start the job and then listen on an event stream for updates. Since we're doing synchronous calls right now, SSE isn't necessary – the request completes when the job is done.

However, if the system already has SSE set up for other parts (for instance, auto-refreshing some stats), it's possible to integrate that. A simple improvement could be to send an SSE event when the report or summary is finished, but this is redundant if we are directly waiting for the response.

Plan: Stick with a **spinner and final alert** approach for now, as recommended [10]. This means: - Show spinner on button click. - If the request fails (HTTP error), hide spinner and show an alert with the error (as done in the JS above). - If the request succeeds, hide spinner and then either auto-show the result or make the link available. You can also briefly flash a success message like " Report generated!" next to the button, which fades out after a few seconds.

This straightforward feedback loop will keep the user informed. It's also suggested in the docs that for now, synchronous processing with just completion feedback is sufficient [10]. The admin knows to wait a bit for the result.

- **Edge cases:** If a script were to take extremely long (minutes), the user might wonder if it's stuck. Without SSE, we can't update progress, but you could update the spinner text like "Still working, please wait..." after a timeout. Another approach is to set a maximum wait time (client-side) and then inform the user that it's taking longer than expected, though ultimately they'd still have to wait or check logs. Since we expect most runs to complete in under, say, 30 seconds, a spinner is fine. If any operation tends to exceed FastAPI's default timeout or CDN timeout, consider optimizing the script or offloading it.

- **Implementing the Spinner:** If not already present, you can implement a basic one:

```
<style>
 .spinner { display: inline-block; width: 16px; height: 16px; border: 2px solid #ccc; border-top:
2px solid #333; border-radius: 50%; animation: spin 0.6s linear infinite; }
 @keyframes spin { to { transform: rotate(360deg); } }
</style>
<span id="spinner" class="spinner" style="display:none;"></span>
```

And in JS:

```
function showSpinner() { document.getElementById('spinner').style.display = 'inline-block'; }
function hideSpinner() { document.getElementById('spinner').style.display = 'none'; }
```

This will show a little loading circle.

Alternatively, use a simple text message:

```
<span id="status-msg" style="display:none; font-style: italic;">Processing...</span>
```

and show/hide that.

- **Error feedback:** As mentioned, if the API returns an error (HTTP 4xx or 5xx), catch it and display an alert or a visible error message on the page. Possibly include the error detail from the server. The docs emphasize to handle unexpected errors (like script failures) gracefully in the UI [10]. For

example, if `data.detail` contains "Report generation failed because ...", show that in the alert so the admin can report it. Also ensure the error is logged server-side.

In summary, **the user experience should be:** Click button -> button disables and spinner shows -> after a few seconds, spinner hides -> either a download starts or a link appears (or a modal opens) -> if error, an alert pops up instead, and maybe the console log has more info. This provides clear feedback and avoids the user clicking multiple times out of confusion [10] .

7. Testing Procedures

Thorough testing is essential to verify that each feature works as expected and that we haven't introduced regressions. Here's a step-by-step test plan:

Unit/Integration Tests for Endpoints: 1. **Local API Tests:** Using FastAPI's TestClient or curl: - **/api/report:** Prepare a `selected_articles.json` with known dummy data (a couple of articles marked approved). Call the endpoint:

```
curl -H "Authorization: Bearer <TOKEN>" -X POST http://localhost:8000/api/report
```

Expect a JSON response containing a `report_path`. Verify that: - The response status is 200 OK. - The JSON has the file path and that file actually exists on disk. - Open the generated Markdown file and check its content. It should contain the expected title (date/keyword) and list the dummy articles with their details (title, summary, etc.). Ensure the content includes all approved articles and no others, and that formatting is correct (e.g., no broken Markdown syntax). - If you run the same request again without changing `selected_articles.json`, check that the file is either the same or overwritten (no duplicate entries). Ideally, the content remains the same – demonstrating idempotence.

- **/api/enrich/keyword:** Set up a test scenario where `selected_keyword_articles.json` or the internal data source has a few articles (both approved and not). You might simulate or stub the summarization logic for testing (since you may not actually call an external API in a test). Call:

  ```
  curl -X POST http://localhost:8000/api/enrich/keyword
  ```

  (with token if needed). Verify:

  - Response 200 with a file path (e.g., `archive/enriched/testkeyword_<date>_all.json`).
  - The file is created. Open it to check it contains summaries for all articles. If you didn't integrate a real summarizer for tests, you might have the script just copy titles into the summary field or some placeholder. That's fine; just ensure the structure is correct and each article is present exactly once.
  - Run the endpoint again and confirm the file is overwritten or unchanged in size (not doubled).
  - Try an edge case: if there are no articles collected (empty list scenario), the endpoint should handle it gracefully (maybe create an empty file or return an error? Ideally, it would return a path to a file that might just contain a message like "No articles to summarize."). Make sure it doesn't crash. If empty, possibly skip file creation and return a message.

- **/api/enrich/selection:** Similar to above but ensure only approved articles are summarized. For testing, mark some articles approved and others not in the data, then call. Verify the output file contains only the approved ones. If the approved list is empty, again handle that (should perhaps return 200 with an empty result or a clear message). No duplicates, and correct handling of consecutive calls.

- **/api/export/md:** Ensure `selected_articles.json` has some content. Call:

```
curl -X GET http://localhost:8000/api/export/md -o out.md
```

  Check:

  - Status 200 and the content is a valid Markdown (open `out.md` in a text editor). The content should include all final selected articles. Verify the fields: title, summary, link, comment, etc., are all present as columns or list items [11] . If an article has no editor_note, it should still be handled (maybe just blank or omitted line).
  - The formatting is consistent for each article. If you had 5 articles selected, you should see 5 entries in the MD.
  - If possible, open the MD in a Markdown viewer or just ensure it's human-readable (e.g. no raw JSON in it).
  - Test idempotence: add a new approved article to `selected_articles.json` and call the endpoint again. The new MD should include the new article (so the output changes appropriately when data changes). Remove duplicates in input and see output remains unique.

- **/api/export/csv:** Similar procedure:

```
curl -X GET http://localhost:8000/api/export/csv -o out.csv
```

  Then:

  - Open `out.csv` in a text editor (or Excel). In a text editor, verify the very first character is the BOM (often shown as "ï»¿" in some editors if not recognized, or you might need a hex editor to see it). In Excel, just check that when opening, the titles and any non-ASCII characters (like Korean) look correct (not gibberish). The presence of BOM should make Excel interpret it in UTF-8 [12] .
  - Check that the CSV columns align, and rows equal the number of selected articles. Every selected article should be a row, with no extra or missing fields.
  - Test without token if you intend to allow direct downloads – if the endpoint is not secured, try in a browser by hitting the URL to see if download triggers.
  - If using token, test that calling without the token returns 401 Unauthorized (the response should ideally include a clear message). This ensures security is working.

- **Security tests:** As noted, ensure that for all POST endpoints and possibly GET exports, if the admin token is missing or wrong, the server responds with 401/403 and no action is taken [2] . This was part of the self-check list [2] . You can simulate this with curl by omitting the header. Also test CORS (if relevant) – if the frontend is served from the same domain, it's fine, but if not, ensure the FastAPI has been configured to allow the origin of the admin interface.

- **Frontend Functionality Tests (Manual UI test):** Open the admin UI in a browser (likely by running `uvicorn` and accessing `index.html`). Manually perform the following:

- Click "일일 보고서 생성". The spinner should appear. After it finishes, verify that a link or button to open the report appears [23]. Click that link: it should open the Markdown report either in a new tab or prompt download. Check the content displayed matches the expected report file that was generated. Also ensure the spinner went away and the button re-enabled. Try clicking it again to see that it still works (possibly overriding the previous file).
- Click the "카드 요약/번역 (전체)" button. Similar process: wait for completion, then click the result link. It should show the summary text file. Verify that the summaries are present and formatted nicely for multiple articles (maybe separated by bullet points or headers). If the content is in JSON and just downloads, consider switching to a more readable format or at least confirm JSON opens. The requirement was to make it likely a text that can be read [19], so if it's not easily readable, that might be a UX issue to note.
- Click "카드 요약/번역 (선정)". Verify it only includes the selected items. If you had, say, 5 approved out of 10 total, make sure only those 5 are summarized. Check the UI doesn't freeze during the wait (the spinner indicates work; the rest of the page should still be browseable, though with the button disabled).
- Click "Export MD". It should immediately download a file (check your browser's downloads). Open that file and confirm content.
- Click "Export CSV". Ensure a CSV downloads. Open it in Excel or a similar program to verify correct encoding and columns. The Excel test is important because the BOM issue is specifically for Excel compatibility [12].
- Try a sequence: generate a report, then immediately export CSV, etc., to see if any interference (there shouldn't be, but it's good to check concurrency, e.g., clicking export while a report generation is ongoing – ideally the UI prevents this by one spinner at a time).
- Try an error scenario: You might force an error by, for example, temporarily renaming the `make_daily_report.py` so the subprocess fails, or simulating an exception in it. Then click the button and observe that an error alert is shown to the user as expected. The spinner should hide after that. Check that any partially created file is handled (maybe it didn't create one at all if failed).

- Test idempotency via UI: Click "일일 보고서 생성" twice in a row (after first is done, click again). Confirm that you don't end up with two different report files for the same day. Ideally, the second click just regenerates the same report (maybe updating the timestamp inside if it prints generation time, but file name would be the same, hence it overwrote). The link might still point to the same path as before, which is good. For summaries, do the same – run twice and ensure the link (and file content) remains one per day/keyword, not growing. If your implementation appends a number to the file name each time, that could cause a new link each time, which might clutter the UI. It's better to consistently use one file name. (One technique: include the date and maybe an internal static "run id" – but since each day is likely only run once, date suffices. If multiple runs per day are likely, you could include a timestamp or simply override.)

- **Automated Testing (if applicable):** The documentation suggests writing Pytest tests for these endpoints [25]. You can automate many of the above checks:

- Use Pytest to spin up the FastAPI TestClient.
- Pre-load some JSON files (perhaps create temp copies of `selected_articles.json` with test data).
- Call the endpoints and assert on responses and file outputs. You might incorporate small files in the repo as test fixtures to compare expected output.

- Test the failure modes by mocking the script functions to throw exceptions, and ensure the HTTP status and message bubble up properly.

Additionally, integration tests that simulate the whole flow (collecting -> approving -> report -> export) were recommended [26]. For example, a test could populate some dummy articles, mark some approved, then call `/api/report`, `/api/export/csv` sequentially, and verify the final CSV contains the same articles that were approved and included in the report [27]. This kind of scenario test ensures the pieces work together.

1. **UI Testing:** While a full Selenium automated test might be beyond scope, consider at least manually testing on multiple browsers. Ensure downloads initiate properly in Chrome, Firefox, etc., and that opening the Markdown doesn't have any styling issues. If the admin UI has both light/dark themes, make sure the inserted elements (buttons, links) look okay in both (colors, visibility). If necessary, adjust CSS.

Test the config and environment: - Try changing a config value, like the archive path, to see if the code respects it. For example, set `"reports": "archive/myreports"` in config.json and run the report generation – does it create the file in the new location? This ensures we didn't hardcode "archive/reports". If it fails, adjust the code to use config. - If the summarization feature depends on an API key, test the two scenarios: - With a dummy key (the script might just pretend to translate or call a fake API). - With no key (the script should use a fallback method per design [7]). For instance, if no translation API is available, perhaps it just copies the English summary or performs a basic truncation. Ensure the script handles that without crashing and maybe logs a warning. You can simulate missing key by not setting the env var and see if `enrich_cards.py` still runs (likely it will do a rule-based summary as per design [7]).

All these tests should pass before considering deployment. The documentation even recommends integrating these into a CI pipeline (GitHub Actions) so that any future changes will run these tests automatically [25] [27]. While setting up CI is beyond this immediate task, keep the tests handy.

8. Self-Checks and Troubleshooting Tips

Before deploying the new features, run through this checklist to ensure everything is in order:

- **Environment Variables:** Verify that all necessary environment variables are properly configured and loaded in the runtime environment [28]. This includes:
- **ADMIN_TOKEN** (or whatever token mechanism is in use) – the server should be reading this from `.env` or deployment secrets. Check the server startup logs or Cloud Run console to ensure the value is set [29]. If the token is missing or mismatched, your endpoints will reject requests erroneously.
- **API keys for external services** – e.g., if using OpenAI API for translation, ensure `OPENAI_API_KEY` (or a generic `API_KEY`) is set in the environment where FastAPI runs. Without it, the summarization script might default to fallback or fail. Check logs for any warning like "No API key provided, using fallback summarization".

- Any other env (database URL if any, although the system currently uses JSON files). If later moving to a DB for selections, ensure connection string is set and the server can connect at startup.

- **Config JSON alignment:** Ensure that `config.json` has entries matching what the code expects. Particularly check:

- `paths.reports` – should be the directory where you want reports. If you changed it, did you also move existing files? The code creates the directory if not exists, but on Cloud Run, ensure the container has write access to that path. Cloud Run file system is ephemeral but writable at runtime – just note that if the container restarts, the archive might be lost unless you have persistent storage. If persistence is needed, consider integrating cloud storage, but that's beyond current scope.
- `paths.selection_file` – should point to `selected_articles.json`. If for some reason it's in a different folder, update config or code accordingly.
- If you decided to use an `enriched` path, consider adding `"enriched": "archive/enriched"` to config.json for consistency. Not strictly required, but it centralizes path settings.
- `features.output_formats` – ensure "md" and "csv" are listed, which they are by default [17]. This is more a note that those formats are intended to be supported.

- If there are any feature flags like `require_editor_approval` (which is true in config) that might affect how selection works, keep them as intended (true means the system only publishes approved articles, which aligns with our usage of selected_articles.json).

- **File permissions and existence:** On the server, verify that the `archive/` directory structure exists and is writable:

- `archive/reports/` – should be present or the code should create it. Our code uses `os.makedirs(..., exist_ok=True)` when writing files, so it should create it if not there.
- `archive/enriched/` – create this directory manually or let the code create it on first run of enrich. Check after first summarization that the directory and file are created. If the code didn't create it, you may need to adjust to ensure `os.makedirs` is called for this path as well.
- The JSON files (`selected_articles.json`, etc.) should be present in the expected location. If any are missing, the scripts might fail reading them. For example, if `selected_articles.json` doesn't exist yet (no selection done), `make_daily_report.py` might throw an error. Ideally, the script handles that by creating an empty file or warning. As a precaution, you might create an empty structure for it:

```
{ "date": "", "articles": [] }
```

  to avoid JSON decode errors. Similarly for `selected_keyword_articles.json` if that is used.

- Check for **data integrity**: If using JSON, sometimes merging manual edits or crashes can corrupt JSON. Use a linter or simply open them to ensure valid JSON format. The docs mention a `repair_selection_files.py` utility to fix any structural issues [30]. If weird behavior happens (e.g., missing articles in output), verify the JSON content matches expectations (no duplicates unless intended, all required fields present). Repair or reset as needed.

- **Logging and error observation:** Keep an eye on server logs while using the features:

- On Cloud Run or wherever deployed, logs should show entries like "Report generated at ..." or any errors in stack traces if something goes wrong. If a feature silently fails, the logs are your first place to look.
- If the UI shows "Report generation failed" error, the server log should have captured why. Common issues could be file I/O errors, key errors if JSON structure unexpected, or exceptions in the scripts.

- Add extra logging around the subprocess calls to log their stderr. For example, if `enrich_cards.py` fails due to an API error, logging `proc.stderr` can help you see it. You might also implement try/except in the scripts themselves to log to a file (maybe `logs/` directory).

- **Token and CORS behavior:** After deployment, test that the admin UI (which might be served as static files or via FastAPI) can actually call these endpoints:

- If the UI is hosted on the same domain, CORS isn't an issue. If not, ensure CORS allows the origin. The initial project likely already set CORS policies in FastAPI. If not, configure FastAPI's `CORSMiddleware` to allow your admin UI origin.
- Confirm that the Authorization header is being sent. In a browser context, if using fetch with a Bearer token, you might need to ensure `mode: 'cors'` and proper headers. If using cookies for auth, ensure `credentials: 'include'` and appropriate cookie flags. All this should be consistent with how other admin API calls (like the gate slider or status refresh) are done.

- As a quick test, open the browser devtools Network tab and trigger one of the new actions. Check that the request is not blocked by CORS and that it includes the auth header. Also check the response status and any message. This can quickly highlight if, say, you forgot to add the Depends(verify_admin) or if the token is wrong.

- **Double-check idempotency in multi-user scenario:** Although this admin is likely single-user or a small team scenario, consider if two people pressed the button around the same time. The second call might come while the first is running. Our implementation is synchronous per request, but FastAPI will handle them concurrently (on different threads). This could potentially start two instances of the script. To guard against that, you might:

- Implement a simple lock mechanism (e.g., using a file lock or threading.Lock in the app) to ensure only one report generates at a time. Given the small scale, this might be overkill, but it's a consideration. Otherwise, two processes might try to write to the same file simultaneously. The window for that is small, but not impossible.
- At least log a warning if a process is already running. A crude approach: create a temp lock file (like `report.lock`) before starting, and if exists, either wait or reject the second request with a message "Another report generation is in progress." This ensures no race conditions for writing the file. If implementing, remember to remove the lock file after done (in finally block).

- Similarly for summarization. But again, this is an edge case; often one admin triggers at a time.

- **Memory and performance:** Running subprocesses will consume memory and CPU. For a small number of articles this is fine. If the collected articles list is huge (say hundreds), summarization could be heavy (especially with external API calls). Monitor the performance:

- Check that the subprocess completes within reasonable time (Cloud Run might have a request timeout around 300 seconds by default; ensure your tasks finish before that). If not, you may need to implement them as background tasks or increase timeouts.

- Ensure that reading large JSON (if lists are long) doesn't break the memory. Python can handle quite large JSON in memory, so likely not an issue unless thousands of entries.

- **Deployment-specific:**

- In Docker/Cloud Run, ensure the `tools/` directory is included in the image and accessible. The scripts should be in the working directory or adjust paths to where they are in the container.
- If Cloud Run is used, files written to the container's filesystem will not persist beyond the life of that instance. That means your `archive/reports` and `archive/enriched` will not be a permanent store if the container restarts or scales. This might be acceptable if the report is mainly for immediate use and archival is manual. But if not, consider integrating a cloud storage (e.g., upload the report to Cloud Storage and return a URL). This is an advanced consideration; for now, just be aware that the "archive" is ephemeral in typical containerized deployments. In a local or VM setup, it would persist.

- If multiple instances of the service run (scaling), each has its own filesystem. Two different instances might not see each other's files. This affects how the user accesses the file via the returned path. You might need to serve the file through an API endpoint rather than direct file path to ensure it comes from the instance that created it, or use a shared storage. For now, if using one instance, this is fine.

- **Troubleshooting tips for issues:**

- Report not generating or empty: Check that `selected_articles.json` has the expected content (approved articles). If it's empty, the script might quickly create an empty report or none at all. The solution is to ensure the selection process is completed before generating report. If the report is partially formatted or missing pieces, compare with a known good output – perhaps some field was missing in JSON and caused the script to skip that part. Update the script to handle missing fields gracefully (e.g., if an article has no summary, maybe leave it blank rather than failing).
- Summary file missing or incomplete: If the summarization finished but no file is found, the script might have not saved it where we expected. Search the project for any default path in `enrich_cards.py`. It might be writing to a different location or naming differently. Align that with what the API expects. Possibly it returns the content directly rather than a file; if so, adapt your endpoint to capture that output and then you could save it to a file yourself or return it directly. If the content is very large, saving to file is fine.
- Download issues: If clicking export doesn't download, check the browser console for CORS or CSP issues. Ensure the `download` attribute is not blocked and that the blob approach is correct. If the file downloads but encoding is wrong, ensure BOM is actually present (open the CSV in a text editor that can show BOM as "EF BB BF" bytes at start).
- UI layout problems: If the new buttons disturb the layout (e.g., wrapping oddly or pushing other elements), adjust with CSS (maybe put them in their own section or use `<br>` as needed). The admin UI might be minimal, but ensure on smaller screens the buttons are still accessible. If the UI uses a framework (Bootstrap, etc.), use its grid system or classes to place your elements.
- Check fallback behavior: The docs mention that currently the UI will fall back to these sync endpoints if async ones are not present [31]. That means perhaps the frontend code might already attempt to call something like `/api/tasks/report` and then default to `/api/report` if not found. Check if any such logic exists in the JS (search for "api/tasks" in the code). If it does, make sure our endpoints work with that logic. Possibly no changes needed, but just be aware.

Finally, perform a **self-check after deployment**: - Open the admin on production/staging, run through each new button as a final acceptance test. - Check that nothing else broke (e.g., existing functions like approving an article, or the gate slider, still work – just to be safe, since we touched adjacent code). - Monitor logs for any errors that didn't show up in testing.

By following these troubleshooting tips and checks, you should catch most integration issues early and ensure the new features run smoothly in production.

9. Final Deployment and Configuration

With implementation and testing done, prepare for deployment. This involves updating environment settings and linking configuration:

- **Update `.env` and Secrets:** If new environment variables were introduced for these features, add them to your `.env` (and to your deployment environment variables).
- For example, if using an `OPENAI_API_KEY` for summarization, ensure it's present in the `.env` file (or provided via your cloud's secret manager). The `.env.example` should be updated to include any new keys for clarity.
- Make sure the ADMIN_TOKEN in `.env` matches what the frontend uses. If you changed it during development, update the production one as well. The admin UI likely uses this token for all admin API calls, so consistency is key.

- If deploying to Cloud Run or similar, check that your CI/CD pipeline or deploy scripts include the latest `.env` values. According to the documentation, Cloud Run environment variables were being configured, so verify on the Cloud Run console that variables like `ADMIN_TOKEN`, `DB_URL` (if any), etc., are correct [28].

- **Include `config.json` in build:** The config file is critical for paths and settings. Ensure it is copied into the Docker image or present at the expected location. If your code looks for it in the working directory, the Dockerfile should COPY it. After deployment, double-check by calling an existing endpoint (maybe `/api/config` if one exists, or just see in logs) that the app loaded the config. If the app falls back to DEFAULT_CFG because it cannot find `config.json`, then your paths might be default (which could still work if defaults are same, but if any differences, it could cause confusion).

- One way to verify is to call an endpoint like `/api/config` (if implemented for gate slider etc.) which returns config values, to see if they match the file. If not, you might need to specify the path or working directory correctly so that `config.json` is read.

- Additionally, if you modified config (like adding "enriched" path), make sure those changes are deployed.

- **Documentation and README:** Update the project's README or admin guide to include usage of the new endpoints [32] [33]. Describe:

- The purpose of each new endpoint (report, enrich, export).
- How to call them and what responses look like (for API documentation completeness, maybe provide example request/response in the README).
- Any config toggles or environment needs (like needing an API key for translation).
- The fact that results are stored in `archive/reports` and `archive/enriched` and are accessible via the returned paths or directly on the server.

- Also update any **admin user manual**: instruct editors how to use the new buttons in the UI, what each does, and any prerequisites (e.g., "Make sure to approve at least X articles before generating a report", or "The summary will be generated in Korean automatically").

- **Final sanity checks:**

- Confirm that the deployment target (server or container) has the same file paths as your dev environment. If not, adjust config. For example, if in Docker the working dir is `/app`, ensure config paths are either relative or adjusted accordingly.
- If using Docker, rebuild the image with the new code and run the container to ensure everything is packaged. Check that `tools/make_daily_report.py` etc., ended up in the image (sometimes missing a `.dockerignore` entry).
- Ensure that the server still starts up without errors. The new imports or scripts might require additional Python packages (for example, if `enrich_cards.py` uses the OpenAI SDK, make sure it's in `requirements.txt`). The `.env` and `requirements.txt` should be updated if needed (the PDFs mention updating requirements is important if new dependencies were introduced [31]).
- After deploying, run through the features one more time in the production-like environment.

Once everything is confirmed, the QualiJournal Admin system will have the new capabilities of daily report archiving, automated card summaries/translations, and one-click exports – all functioning reliably and without duplication issues [10] [34]. This closes the loop on the mid-term plan features and provides a smoother workflow for the editors, who can now collect, approve, summarize, publish, and export news all within one system [35] [11].

---

[1] [3] [4] [24] 1010_1_나머지 계획_쿼리저널 관리자 시스템 개선 작업 인수인계 보고서.pdf
file://file-XBYbxoGjLn4pSrN6QHLGAd

[2] [5] [6] [8] [9] [10] [11] [12] [15] [18] [19] [20] [21] [22] [23] [25] [26] [27] [28] [29] [31] [32] [33] [34] 1011_1쿼리저널 관리자 시스템 (Admin API) 개선 작업 인수인계 보고서.pdf
file://file-MeqUbXwLc5KBspHeGiMAGN

[7] 1003_3A 하루 한 키워드" 역사 큐레이션형 퀄리뉴스를 구축하기 위한 구체적 개발 가이드입니다.pdf
file://file-92Q3asQNjsGDWWy73Fkbpt

[13] [14] [16] [17] orchestrator.py
file://file_0000000059a86246a59c9c3d2d40cd3f

[30] 1005_1A_개선방안report.pdf
file://file-HzdZ5Qgkfa9y6poNMdPG28

[35] 1003퀄리뉴스 _키워드 뉴스_ 기능 설계 및 구현 전략.pdf
file://file-XoVMv2VGBLyHnFVetrmoGx