ChatGPT

Plan for Automated Monitoring of New English-Taught Degree Programmes in Germany

1. Data Sources and Extraction Strategy

To capture **new English-taught degree programmes** at public German universities, we will aggregate data from multiple authoritative public sources:

- **HRK Hochschulkompass (Higher Education Compass):** The HRK's database lists all study programs in Germany (mostly public or officially recognized institutions) [1]. We will use its advanced search to filter for **English as the main language of instruction** and restrict results to **public universities** [1]. The Hochschulkompass provides detailed programme info (degree type, institution, language, etc.), directly submitted by universities, making it a key source. *(If an API or export is not available, the ETL script will perform an HTTP GET or HTML form simulation to retrieve search results pages for English-taught programs.)*

- **DAAD International Programmes Finder:** The DAAD maintains a database of *"internationally oriented study programmes offered by German universities and research institutions"* [2], which essentially highlights programmes geared towards foreign students (typically English-taught). We will query this for English-taught Bachelor's, Master's, and PhD programmes. The DAAD site allows filtering by language and degree level [3]. It often includes notes on tuition fees (many listed programmes are **tuition-free aside from semester fees** [4]). Since DAAD does **not provide an official API**, our ETL will rely on web scraping techniques to parse the programme listings [5].

- **German Accreditation Council Database:** The **Akkreditierungsrat** database contains all accredited study programmes in Germany (since Jan 2019) [6]. This can help catch new programmes as they receive accreditation. We will attempt to query this database (note: the interface is German-only and possibly a dynamic web app [7]). A strategy here is to find newly accredited programmes by filtering on accreditation date or by periodically scraping all accredited programmes and extracting those not seen before. (If direct queries are difficult, the script might use an automated browser or reverse-engineer the web requests that the search interface uses.) *Be mindful that the Accreditation Council warns of potential data quality limitations after their data migration [6], so some entries may have inconsistencies.*

Data Fields to Extract: From each source, we will parse key fields for each degree programme, such as: - **Institution Name** (and type/status, e.g. university vs. FH, and whether public), - **Programme Name** (and degree level, e.g. B.Sc., M.A.), - **Language of instruction** (to ensure it's English or multi-lingual including English), - **Tuition Fees** (amount per semester or indication of "tuition-free"), - **Start Date or Next Intake** (e.g. Winter/Summer semester start), - **Link/Reference** (URL to the source detail page for more info), - **Last updated or accreditation date** (if available, to help identify newness).

Each source may have slightly different data; the ETL will normalize this into a common structure. For example, tuition might be given as "None" or "0 EUR" or "-" which we'll standardize to numeric 0 for free programmes. Institution names from different sources will be matched (we may maintain a mapping or use string similarity to unify names).

Approach: The Python ETL scripts will use HTTP requests to fetch data. For pages that are HTML, we'll use **BeautifulSoup** to parse the content. For any JSON/XML endpoints (if discovered via network calls or provided), we'll use those directly for reliability. If a source is highly dynamic (e.g., the accreditation database likely uses a web application), we may use a headless browser (e.g. Selenium or Playwright) or analyze the network calls to call the same REST endpoints the site's frontend uses.

Each source's extraction can be a separate module or script, writing results to a common staging area (like CSV or directly to the database). We will implement **filters in the extraction** (where supported by the source) to limit data to *English-taught programmes at public institutions* – this reduces noise and load. For instance, Hochschulkompass allows filtering by main language and institution type in the query itself [1], and the DAAD finder allows filtering to English language [3]. If needed, additional filtering can be done in Python after retrieving a broader dataset.

2. Data Storage Design (PostgreSQL Database)

We will set up a **PostgreSQL** instance (likely as a Docker service in our compose stack) to store the aggregated programme data. A relational database is ideal here because we have structured, relational data and will query it frequently with complex filters (language, date, tuition) – a use case for which SQL databases are well suited [8]. Key tables might include:

- **institutions** – storing each university/college with fields: `inst_id` (PK), name, type (e.g. "Universität" vs "FH"), status (public or private), location (state or city), etc. This helps normalize institution names and mark which are public. We can initially load known public institutions (from HRK or an official list) so we have a reference. This table is relatively static (changes only when new institutions appear).

- **programmes** – storing degree programme entries with fields: `prog_id` (PK), `inst_id` (FK to institutions), program name, degree (e.g. MSc, BA), language(s), tuition_fee, start_date (or next intake info), source (HRK, DAAD, GAC), and timestamps (`created_at` when first seen, `updated_at` on changes). We also include a flag or date for accreditation if available. The `created_at` timestamp will be crucial for identifying "new" programmes in the last 24h.

- (Optional) **programme_history** or a changelog – if we want to track changes or removals. But initially, we can keep it simple: update in place and perhaps mark a program as inactive if it disappears from sources for a while.

Docker Compose Setup: In our Docker Compose file, we will add a Postgres service (with a volume for persistence). Node-RED (already running in Docker) will be configured to connect to this DB for queries. Ensure the Compose uses a network so Node-RED can reach the Postgres service by name. We'll also expose Postgres on a port for the ETL scripts (if running directly on the host or via cron) to connect.

3. Python ETL Pipeline (Daily Data Refresh)

We will create Python-based ETL scripts to **Extract, Transform, and Load** the data into PostgreSQL, running on a daily schedule:

3.1. ETL Script Structure

Each run of the ETL will perform roughly the following steps:

1. **Extract from Sources:** Call each data source:
2. For **HRK Hochschulkompass**: Issue an HTTP GET with query parameters (or POST form) to retrieve all English-taught programmes at public institutions. Parse the returned HTML for the list of programmes (likely paginated, so handle multiple pages if needed). For each programme entry, extract the fields (name, degree, institution, etc.), and follow the detail page if necessary to get full details like tuition or start date.
3. For **DAAD International Programmes**: Request the search results for English-taught programmes. DAAD's interface might paginate as well. Extract similar fields. (If the DAAD "All study programmes in Germany" page is easier to scrape, we might use that, as it could mirror the HRK data [3], but focusing on the "International Programmes" ensures we catch those targeted to international students.)

4. For **Accreditation Council (GAC)**: Either scrape for newly accredited programmes or retrieve all and filter by accreditation date. Possibly use an "advanced search" to filter by **language = English** if their interface allows (or search keywords like "Englisch" in program names/description). Extract programme name, institution, degree, accreditation date, etc.

5. **Transform:** Normalize and clean the data:

6. Ensure consistent language notation (e.g., "English" vs "Englisch" all set to a standard value).
7. Convert tuition fee strings to a numeric value (e.g., "None" or "no tuition" -> 0). We'll also capture the currency if any (most are in EUR) and period (per semester/year).
8. Standardize date formats (e.g., start dates or accreditation dates to `YYYY-MM-DD`).
9. Map institution names to our `institutions` table entries. If an institution is new (not in DB), insert it with its attributes (and mark if public – for HRK, we know it's public; for DAAD, we may need to cross-check with our institutions list or assume if it appears in DAAD list, it's recognized).

10. Prepare a list of programme records for loading. If multiple sources list the same programme, we should merge them (e.g., ensure we don't create duplicates). A strategy is to use a composite key of `(institution, program_name, degree)` to identify uniqueness. If a programme appears in HRK and DAAD, it should end up as one entry in the DB (we might update it to combine info, e.g., if one source had missing tuition info but another provided it).

11. **Load into PostgreSQL:** Using a library like `psycopg2` or SQLAlchemy, connect to the Postgres DB and upsert the data:

12. **Insert new programmes:** For each programme not already in the DB (by unique key), insert a new row with `created_at = now()`. This will mark it as a **new programme** that can trigger notifications.
13. **Update existing programmes:** If we find an existing entry (matching key) but some details have changed (e.g. tuition fee updated or an additional language), update the record (`updated_at = now()`). We might not treat updates as "new" for notifications, unless we want to notify changes – but the focus here is on new programmes.
14. We may implement this upsert logic either via SQL `INSERT ... ON CONFLICT` (with unique constraint on the composite key) or by selecting existing IDs first then deciding insert/update in Python.

15. Optionally, handle programmes that were present previously but no longer appear (could mark them as inactive or remove). However, for detecting new programmes, removal isn't a primary concern. (We might simply log if something disappeared, for manual review.)

16. **Logging and Verification:** The script will log its actions (e.g., "10 new programmes inserted, 1 updated, 0 removed"). If possible, we'll capture any errors and either send an alert or write to a log file for debugging.

3.2. Scheduling the ETL

To run this daily, we have two main approaches:

- **Cron Job on the Ubuntu VM:** Set up a cron entry (e.g., in crontab) to run the ETL script every night (e.g., 2:00 AM server time). This could invoke a Python script directly or a Docker container. For example, using Docker Compose, we might include a service for the ETL that runs once and exits; combined with a cron task (`docker compose run etl-container`) nightly. Cron ensures the job runs even if Node-RED is running separately.

- **GitHub Actions (CI):** Alternatively, host the ETL script in a GitHub repo and use a scheduled GitHub Action to run it daily. The action would connect to the live Postgres database to perform the update. (This requires the Postgres instance be accessible or using an SSH runner.) This approach offloads the scheduling to GitHub's infrastructure and can send notifications if the workflow fails. **Note:** Using GH Actions means storing DB credentials as secrets and potentially opening a port; if the VM is not publicly accessible, cron is simpler.

In either case, after scheduling, **verify the automation**: the ETL should be populating new data daily without manual intervention. We will also implement safeguards like not running multiple ETLs concurrently (to avoid race conditions).

4. Node-RED Flow Configuration for Daily Monitoring

Node-RED will orchestrate the detection of new entries and trigger notifications + dashboard updates. We will create a dedicated flow for this:

4.1. Database Query Trigger: Use a **schedule trigger node** in Node-RED to run once every day (e.g., an Inject node configured for a specific time each morning, or the `cron-plus` node for flexible scheduling).

This trigger will initiate the check for new programmes. (If the ETL is at 2 AM, we might trigger Node-RED at 3 AM or 8 AM to ensure fresh data is in the DB.)

4.2. Query New Programmes: Add a **PostgreSQL node** (via `node-red-contrib-postgresql` or similar) in the flow, which executes a SQL query on the Postgres database [9]. The query will select programmes added in the last 24 hours, filtered by the desired criteria. For example:

```
SELECT p.program_name, i.name AS institution, p.degree, p.language,
p.tuition_fee, p.start_date
FROM programmes p
JOIN institutions i ON p.inst_id = i.inst_id
WHERE p.created_at >= NOW() - INTERVAL '24 hours'
  AND LOWER(p.language) LIKE '%english%'
  AND p.tuition_fee <= 1000;
```

This assumes we treat anything added in the past day as "new". (The tuition_fee filter `<= 1000` is just an example – the user can set a max tuition threshold, e.g. €0 for strictly tuition-free, or some nominal amount to include low semester contributions.) The language filter ensures we only get English-taught entries (if some slipped in with multiple languages, we adjust accordingly).

The Postgres node will return the result rows to Node-RED. **Note:** We ensure the Node-RED DB credentials are configured correctly (host, port, user, password) in the node settings [10].

4.3. Filter/Branch on Results: Add a function or switch node to handle two cases: - If the query returns **no new programmes** (empty result), the flow can simply end or log "No new programmes found today." (We might not send an empty email). - If there **are new programmes**, proceed to notification nodes.

4.4. Compose Notification Messages: Use a Function node to format the output for notifications. For example, construct an email body listing each new programme with its details and source link. The message could look like:

New English-taught Programmes (Last 24h):
- **M.Sc. Data Science** – University of Example (Start: Oct 2025, *tuition-free*)
- **B.Sc. Artificial Intelligence** – Tech University X (Start: Oct 2025, €1500/yr)
(Total: 2 new programmes)

This function can also tailor the message format for Telegram (plain text or Markdown) vs email (HTML or text). Include enough info for the user to decide interest, e.g., programme name, institution, start date, tuition, and maybe a URL for more info (maybe the source link from HRK or DAAD).

4.5. Notification Nodes: Connect the formatting node to multiple outputs: - **Email Notification:** Configure Node-RED's email node (e.g., using SMTP details of an email account or a service) to send an email to the user (and others if needed). The subject could be "New English Programs in Germany - Update". Node-RED's email node will use the message payload as the body. - **Telegram Notification:** If the user prefers instant alerts, use Node-RED's Telegram sender node (`node-red-contrib-telegrambot`). We'll set up a

Telegram Bot API token and chat ID for the user or group. The message payload (from the function) goes to this node to send a Telegram message. (We may shorten the text or send multiple messages if too long, as Telegram might have length limits.)

We can add other channels similarly (SMS, Slack, etc.) if needed in the future by branching off the same function output.

4.6. Node-RED Flow Deployment: We deploy this flow in the Node-RED editor. The inject (schedule) node ensures it runs daily automatically. We can also manually trigger it (inject node supports manual trigger) for testing. The **flow will look like**: *[Inject Timer] → [PostgreSQL Query] → [Function (format & decision)] → [Email] & [Telegram]*.

Node-RED is well-suited here as it can easily interface with databases and send emails/Telegram, all in a visual flow [9] [11]. Using Node-RED also makes it easy to adjust filters or add new notification methods without writing new code.

5. Dashboard and API for Reviewing Results

In addition to push notifications, we will provide a way to **browse or query** the collected data:

- **Node-RED Dashboard:** We will utilize Node-RED's Dashboard nodes (`node-red-dashboard` package) to set up a simple web dashboard. For example, we can create a **table view** showing recent programmes. Using a **UI Table node**, we can display the list of new programmes (with columns for name, institution, etc.) on a dashboard accessible via Node-RED's UI (e.g., at `http://your-vm:1880/ui` by default) [12]. We can also add date pickers or filters on the dashboard for the user to query different time ranges or criteria on demand. This gives a quick visual overview for review.

- **JSON API Endpoint:** For a more programmatic access or if the user wants to integrate with other tools, we can set up a simple **HTTP API** in Node-RED:

- Use an **HTTP In node** (listening on an endpoint like `/api/new-programmes` or `/api/programmes`) that triggers a flow.
- The flow will query the database (similar to the above, but perhaps allowing a query parameter like `since=<date>` or `days=<n>` to get last N days of data).
- Use an **HTTP Response node** to output the data as JSON. Essentially, Node-RED can take the DB result and pass it through without formatting (or format as JSON). The result would be a list of programme objects in JSON that the user (or another app) can fetch via GET request.

This API could be protected or left open on local network as needed. Alternatively, if we anticipate heavy use, we might consider directly using a tool like **PostgREST** to expose the Postgres data as REST API, but given Node-RED is already in place, it can handle this lightweight API need.

The combination of a dashboard and an API means the user can either visually inspect new programmes or pull the data into other applications (for example, into a spreadsheet or a custom UI) as needed.

6. Key Benefits of the System

- **Comprehensive & Timely Coverage:** By combining multiple sources (HRK, DAAD, and accreditation data), the system ensures that *no new English-taught programme slips through unnoticed*. As soon as a new course is listed in these public databases, it will be picked up within a day and logged.

- **Automation = Time Savings:** The entire workflow – from data collection to notification – is automated. This saves the user from manually checking websites or newsletters. The personal VM does the heavy lifting daily, and the user is simply alerted to results.

- **Customized Filtering:** Only programmes meeting the user's criteria are surfaced – e.g. taught in English and under a certain tuition cost. This avoids spam and focuses attention on relevant opportunities. (For instance, the system will exclude overpriced private programs, which are often less reputable [13] [1], and highlight mostly tuition-free public programs, as indeed *"the majority of them are tuition free"* in Germany [4].)

- **Central Data Repository:** Storing the data in PostgreSQL provides a historic record of programmes. The user can perform analysis, see trends (e.g., how many new English programs appear each year, or which universities are launching new courses), and even use SQL queries for custom reports. A relational DB offers powerful querying capabilities for such data [8].

- **Real-Time Alerts:** Instant notifications via email/Telegram mean the user can react quickly (useful if application deadlines are near or competition for spots is high). This proactive alerting is more effective than reactive searching.

- **Transparency and Control:** The solution uses open-source tools (Python, Node-RED, Postgres). The user retains full control – they can modify the ETL logic or Node-RED flows as needed (for example, to add another source or change the filtering criteria). There's no black-box: all data and code reside on the user's VM.

- **Scalability and Extensibility:** The modular design (separate source scrapers, a normalized DB, Node-RED for orchestration) makes it easy to extend. For example, adding another source (like the EU's ECTS database or others) or additional notification channels is straightforward. The system could also be scaled to monitor programmes in other languages or countries by adding sources and adjusting filters, without a complete redesign.

7. Common Pitfalls and Challenges

While the plan is robust, a few challenges and pitfalls need to be addressed:

- **Website Changes & Scraping Fragility:** The scraper scripts may break if the target websites change their layout, URL structure, or require new interactions. Web scraping is not a "write once, run forever" operation – *websites evolve frequently, and failing to monitor your scraper can lead to data quality issues or broken pipelines* [14]. We must be prepared to update the parsing logic when sites update their HTML or if an API becomes available.

- **Rate Limiting & Blocking:** Pulling data daily is generally low-frequency, but if the script is too aggressive (e.g., pulling thousands of records without delay), the source websites might throttle or block our requests. We should respect each site's `robots.txt` and use reasonable delays between requests. If needed, implement rate limiting in the ETL (e.g., small sleep between page fetches) and use a descriptive User-Agent string. Since sources like DAAD have no API [5], we rely on scraping and must do so politely.

- **Data Quality & Consistency:** The data comes from different sources with varying quality. There might be missing fields, inconsistent program names, or duplicates:

- The **Accreditation Council** explicitly notes limitations in data quality due to migration [6] – some records might have typos or incomplete info.
- Different sources might spell an institution's name slightly differently or use different English names, causing duplication if not handled (e.g., "Technical University X" vs "TU X"). We mitigate this by normalizing names and possibly using the accreditation or HRK IDs if available.
- Language of instruction might be multi-valued (e.g., "English/German" programs). Our filter might include those or exclude them depending on criteria. We should decide how to treat such cases (likely include if English is one of the languages).

- Some programmes might appear in both HRK and DAAD; without careful deduping, we could count them twice. Our ETL's merge logic must handle this.

- **Scraping Dynamic Content:** The accreditation database's search appears to be a dynamic web app. If data is not easily accessible via simple requests, we might need to simulate a browser. This introduces complexity (needing a headless browser and more resources) and potential flakiness. We should assess if the effort is justified or if relying on HRK/DAAD covers most new programmes (most public uni programs will be in HRK/DAAD anyway). Perhaps use accreditation data as a secondary check.

- **Cron/Workflow Timing Issues:** If the ETL and the Node-RED query are not synchronized, we might miss or double-count entries. For example, if Node-RED checks at midnight but the ETL hadn't run yet that day, it would find nothing; or if it runs twice in 24h, it might catch the same "new" item twice. We should align schedules and possibly use a marker (like last run time). Keeping it once per 24h consistently will avoid overlap.

- **Notification Noise:** If many programmes appear at once (e.g., if universities add a batch at the start of application season), the email/Telegram might become lengthy or even hit size limits. The user might be overwhelmed. We should consider batching: e.g., if more than X new items, send a summary ("10 new programmes added, see dashboard for details") instead of an extremely long message. Fine-tuning the notification content will be important for long-term use.

- **System Resource Use:** Running daily scrapers and a Postgres DB on a personal VM is usually fine, but if the data grows large (tens of thousands of entries), queries might slow down. We should archive or cleanup old data if not needed. Also, ensure Docker has enough memory (Node-RED and Postgres are not heavy, but headless browser for scraping could be). Monitor disk space for the DB over time.

- **Legal and Ethical Considerations:** All data sources are public, but we should ensure compliance with their terms of use. It's likely acceptable for personal/research use to scrape these. We should avoid overloading their servers or redistributing the data improperly. Also, we are dealing with public university programme info, so no personal data concerns here (thus GDPR etc. are not a big issue, but still good to be mindful of terms of service).

8. Fallback Strategies and Resilience

To make the system robust, we plan for failures and how to handle them:

- **ETL Failure Recovery:** If a daily ETL run fails (due to a site being down, network issue, or a parsing error), implement a few **fallback steps**:
- The ETL script can be configured to *retry* a failed source after a short delay, or try a secondary source (if HRK is down, maybe DAAD still provides partial info).
- Maintain the last successful run's data. If today's run fails entirely, the system can either keep using yesterday's data (with no new entries) or send an alert that the ETL failed. We can integrate a simple check: Node-RED can monitor a "last_updated" value in the database (or a file). If no update occurred in >24h, send an email notification to the admin (us) that "ETL did not run – data may be stale."

- If using GitHub Actions, failures there would be reported in the repo. We could also have the GH Action send an email on failure. On the VM with cron, we can use cron's mailing of output or redirect logs to a file and have Node-RED tail that log for errors.

- **Data Consistency Checks:** Implement sanity checks in ETL: e.g., if an unusually low number of programmes is fetched (perhaps indicating a partial page parse failure), log a warning or do not overwrite the DB blindly. This prevents wiping out data due to a bad scrape. Instead, we might keep the old data and try again later.

- **Duplicate Handling:** In case duplicate notifications slip through (e.g., if the same new programme is picked up from two sources on the same day with slight differences), we can add logic in Node-RED to suppress sending the same program twice. For instance, store a cache of notified programme IDs for 1-2 days. Node-RED could maintain context or a small file with the IDs of programmes it already notified, and filter them out if they appear again (though with proper DB unique constraints and one insertion, this shouldn't happen).

- **System Downtime:** Ensure the Docker services restart on reboot or failure (use `restart: unless-stopped` in compose). That way if the VM restarts, Node-RED and Postgres come back up automatically. The cron or scheduler should also come back (cron will, GH Actions independent of VM). If Node-RED is down during the scheduled time, we might miss a notification – but the data would still be collected. As a backup, one could run a second notification later or on next start. We might also set Node-RED's inject node to `inject once on start` (with a small delay) – so if Node-RED restarts, it will immediately check for any new entries since the last run.

- **Stale Data Alerts:** If, for example, no new programme has appeared for a very long time (which might be normal during off-cycle months, but could also indicate something's wrong with the scraper), we could have a scheduled Node-RED flow to notify "No new programmes have been detected in X days." This would prompt a manual check to ensure the pipeline is still working.

- **Backups:** Regularly back up the PostgreSQL data (the list of programmes). This could be done via a simple `pg_dump` on a weekly basis. While the data can be re-scraped from sources, historical records (especially if a programme gets removed later) might be valuable to keep. Storing backups off-site (or at least in a different volume) protects against data loss.

- **Graceful Degradation:** If one source becomes too difficult to scrape (say the accreditation database), the system should continue using the other sources. Each source's ETL can fail independently without breaking the whole pipeline. We'll design the ETL such that errors on one source don't halt the others. They can log an error and skip, so partial updates still go through. This way, if e.g. HRK is temporarily down, the DAAD data still updates (and vice versa).

By anticipating these issues and having fallbacks (retries, alerts, backups), the monitoring system will remain reliable and maintainable. It's important to periodically review the system logs and update the scrapers as needed to keep the data flow smooth.

9. Conclusion

With this plan, we set up a **fully automated monitoring system** that **daily scans official databases for new English-taught degree programmes in Germany, stores them in a structured way, and alerts the user to any new opportunities**. The use of Docker Compose (for PostgreSQL and Node-RED) and simple scheduling (cron or GitHub Actions) ensures the solution can run 24/7 with minimal oversight. By leveraging multiple data sources and robust filtering, the user can trust that they won't miss any new program that fits their criteria, without sifting through irrelevant information. The combination of email/Telegram notifications and a web dashboard/API provides both immediacy and convenience for reviewing the results. Overall, this system empowers the user to stay informed about Germany's English-taught programmes in a timely and efficient manner, while minimizing manual work and maximizing reliability.

Sources:

- Reddit user recommendation on Hochschulkompass (public universities database & language filters) [1]
- Reddit user recommendation on DAAD database (English-taught programmes, tuition free) [3] [4]
- DAAD API absence and need for scraping [5]
- Accreditation Council database information (data since 2019, quality limitations) [6]
- Node-RED integration with PostgreSQL (example workflow) [9] [11]
- Node-RED Dashboard accessibility [12]
- Best practices for data storage (using databases for large/frequent queries) [8]
- Warning on scraper maintenance (website evolution requires monitoring) [14]

---

[1] [3] [4] [13] What's the deal with English-taught degree programs in Germany? : r/germany
https://www.reddit.com/r/germany/comments/15i9pzv/whats_the_deal_with_englishtaught_degree_programs/

[2] International Programmes - Homepage - DAAD
https://www2.daad.de/deutschland/studienangebote/international-programmes/en/

5 GitHub - theotherstupidguy/daadde-exposed: Daadde Exposed is a Sinatra\Grape\Mongoid App that provides you with an API to use the German Academic Exchange Service https://www.daad.de/en/index.html
https://github.com/theotherstupidguy/daadde-exposed

6 7 Accredited study programmes and higher education institutions | Stiftung Akkreditierungsrat
https://www.akkreditierungsrat.de/en/accredited-institutions-higher-education-and-study-programmes/accredited-study-programmes-and

8 14 Web Scraping Best Practices - Abstract API
https://www.abstractapi.com/guides/other/web-scraping-best-practices-a-comprehensive-guide

9 10 11 12 OSGS - Node-red workflow that accesses data in Postgres
https://kartoza.github.io/osgs/workflows/node-red-access-data-in-postgres.html