IMAGO Coding Challenge C3

Submitted by Alexander Tarnavsky Eitan

Deliverables

The code is available on GitHub: https://github.com/aletar89/imago-challenge with a README here.

The API is deployed at: https://imago-challenge.onrender.com (Please note that the deployed instance will spin down with inactivity, which can delay the first request by about a minute)

Approach

An API for querying an ElasticSearch DB is a standard piece of code with no complex logic so I used Claude to generate the bulk of the code and focused on making the code clear and maintainable.

I chose to emphasise the back-end using a Flask API which also serves a simple vanilla JS front-end for showcasing the results.

For testing, I used mostly integration tests because the functionality is very simple and the unit tests felt a bit useless for this exercise. For a production code I would set a coverage goal and add unit tests for the sections that the integration tests didn't cover sufficiently. I also decided to run the tests against the dev ES server provided and not mocking it. This provides better coverage of real edge cases and the load on the server is not great and the tests still run quickly for this exercise. The downside of this is that if the dev server is different from the production server the tests may be misleading.

Discussion

Data Normalization

To explore the actual unstructured data as it exists in the DB, I created a script that samples items from the DB and shows the types and cardinality of the fields. This revealed that, in contrast to the ES index mapping, the `bearbeitet_bild`, `description` and `title` fields are never present. Thus, in practice, searching is performed only on the `suchtext` field.

In the API, I construct the search query such that it looks for the query term in all relevant fields.

One way to improve searchability is use AI to detect objects and people in the images and include these as tags. Another way is including the description of where the image was used.

Problem identification

The content of the fields in the DB items is user generated so we need to check them for malicious content before sending results to the user's front-end to prevent XSS attacks. To address this issue, I used the bleach library to sanitize the inputs.

Another security issue is that the search query and filters can be used in an ES injection attack. This is prevented by using `multi_match` which doesn't interpret Lucene operators and passing the query as a dict to the official ES python client and not handling the JSON query directly.

Scalability

Large volumes of data would be handled mostly by the ElasticSearch server which has built in solutions for caching, sharding and horizontal scaling.

The volume of data handled by the fetching API is limited by pagination so there shouldn't be any latency or memory issues. The API is scalable, so for a high number of requests the solution is creating many instances of the same pod and placing them behind a load balancer.

Maintainability

To keep the code maintainable for additional media items and external sources I created an abstraction layer with a generic class for a media item and an abstract class for a fetcher with the current ElasticSearch fetcher inheriting from it.

This allows the front-end to stay the same expecting to get a media item while the back-end API can query multiple sources internally using the same signature inherited from a common fetcher class.