How does it work?

> The following text was originally published on the seekseek website, at https://seekseek.org/technology.

> The information on this page is currently changing. While the current deployment of seekseek still does use the technology as described here, a next version is currently being tested which has some significant architectural changes to better achieve the goals stated below, and to be more maintainable in the long term.

The technology

So... what makes SeekSeek tick? Let's get the boring bits out of the way first:

- The whole thing is written in Javascript, end-to-end, including the scraper.
- Both the scraping server and the search frontend server run on NixOS.
- PostgreSQL is used as the database, both for the scraper and the search frontends (there's only one frontend the time of writing).
- The search frontends use React for rendering the UI; server-side where possible, browser-side where necessary.
- Server-side rendering is done with a fork of `express-react-views`.
- *Most* scraping tasks use bhttp as the HTTP client, and cheerio (a 'headless' implementation of the jQuery API) for data extraction.

None of that is really very interesting, but people always ask about it. Let's move on to the interesting bits!

The goal

Before we can talk about the technology, we need to talk about what the technology was built *for*.

SeekSeek is radical software. From the ground up, it was designed to be FOSS, collaborative and community-driven, non-commercial, ad-free, and to improve the world - in the case of SeekSeek specifically, to improve on the poor state of keyword-only searches by providing highly specialized search engines instead!

But... that introduces some unusual requirements:

- **It needs to be resource-conservative:** While it doesn't need to be *perfectly* optimized, it shouldn't require absurd amounts of RAM or CPU power either. It should be possible to run *the whole thing* on a desktop or a cheap server - the usual refrain of "extra servers are cheaper than extra developers", a very popular one in startups, does not apply here.
- **It needs to be easy to spin up for development:** The entire codebase needs to be self-contained as much as reasonably possible, requiring not much more than an `npm install` to get everything in place. No weirdly complex build stacks, no assumptions about how the developer's system is laid out, and things need to be debuggable by someone who has never touched it before. It needs to be possible for *anybody* to hack on it, not just a bunch of core developers.
- **It needs to be easy to deploy and maintain:** It needs to work with commodity software on standard operating systems, including in constrained environments like containers and VPSes. No weird kernel settings, no complex network setup requirements. It needs to Just Work, and to *keep* working with very little maintenance. Upgrades need to be seamless.
- **It needs to be flexible:** Time is still a valuable resource in a collaborative project - unlike a company, we can't assume that someone will be able to spend a working day restructuring the entire codebase. Likewise, fundamental restructuring causes coordination issues across the community, because a FOSS community is not a centralized entity with a manager who decides what happens. That means that the core (extensible) architecture needs to be right *from the start*, and able to adapt to changing circumstances, more so because scraping is involved.
- **It needs to be accessible:** It should be possible for *any* developer to build and contribute to scrapers; not just specialized developers who have spent half their life working on this sort of thing. That means that the API needs to be simple, and there needs to be space for someone to use the tools they are comfortable with.

At the time of writing, there's only a datasheet search engine. However, the long-term goal is for SeekSeek to become a large *collection* of specialized search engines - each one with a tailor-made UI that's ideal for the thing being searched through. So all of the above needs to be satisfied not just for a datasheet search engine, but for a *potentially unlimited* series of search engines, many of which are not even on the roadmap yet!

And well, the very short version is that *none* of the existing options that I've evaluated even came *close* to meeting these requirements. Existing scraping stacks, job queues, and so on tend to very much be designed for corporate environments with tight control over who works on what. That wasn't an option here. So let's talk about what we ended up with instead!

The scraping server

The core component in SeekSeek is the 'scraping server' - an experimental project called srap that was built specifically for SeekSeek; though also designed to be more generically useful. You can think of srap as **a persistent job queue that's optimized for scraping**.

So what does that mean? The basic idea behind srap is that you have a big pile of "items" - each item isn't much more than a unique identifier and some 'initial data' to represent the work to be done. Each item can have zero or more 'tags' assigned, which are just short strings. Crucially, none of these items *do* anything yet - they're really just a mapping from an identifier to some arbitrarily-shaped JSON.

The real work starts with the **scraper configuration**. Even though it's called a 'configuration', it's really more of a *codebase* - you can find the configuration that SeekSeek uses here. You'll notice that it defines a number of tasks and seed items. The seed items are simply inserted automatically if they don't exist yet, and define the 'starting point' for the scraper.

The tasks, however, define what the scraper *does*. Every task represents one specific operation in the scraping process; typically, there will be multiple tasks per source. One to find product categories, one to extract products from a category listing, one to extract data from a product page, and so on. Each of these tasks has its own concurrency settings, as well as a TTL (Time-To-Live) that defines after how long the scraper should revisit it.

Finally, what wires it all together are the *tag mappings*. These define what tasks should be executed for what tags - or more accurately, for all the items that are tagged *with* those tags. Tags associated with items are dynamic, they can be added or removed by any scraping task. This provides a *huge* amount of flexibility, because any task can essentially queue any *other* task, just by giving an item the right tag. The scraping server then makes sure that it lands at the right spot in the queue at the right time - the task itself doesn't need to care about any of that.

Here's a practical example, from the datasheet search tasks:

- The initial seed item for LCSC is tagged as `lcsc:home`.
- The `lcsc:home` tag is defined to trigger the `lcsc:findCategories` task.
- The `lcsc:findCategories` task fetches a list of categories from the source, and creates an item tagged as `lcsc:category` for each.
- The `lcsc:category` tag is then defined to trigger the `lcsc:scrapeCategory` task.
- The `lcsc:scrapeCategory` task (more or less) fetches all the products for a given category, and creates items tagged as `lcsc:product`. Importantly, because the LCSC category listings *already* include the product data we need, these items are immediately created with their full data - there's no separate 'scape product page' task!
- The `lcsc:product` tag is then defined to trigger the `lcsc:normalizeProduct` task.
- The `lcsc:normalizeProduct` task then converts the scraped data to a standardized representation, which is stored with a `result:datasheet` tag. The scraping flows for *other* data sources *also* produce `result:datasheet` items - these are the items that ultimately end up in the search frontend!

One thing that's not mentioned above is that `lcsc:scrapeCategory` doesn't *actually* scrape all of the items for a category - it just scrapes a specific page of them! The initial `lcsc:findCategories` task would have created as many of such 'page tasks' as there are pages to scrape, based on the amount of items a category is said to have.

More interesting, though, is that the scraping flow doesn't *have* to be this unidirectional - if the total amount of pages could only be learned from scraping the first page, it would have been entirely possible for the `lcsc:scrapeCategory` task to create *additional* `lcsc:category` items! The tag-based system makes recursive discovery like this a breeze, and because everything is keyed by a unique identifier and persistent, loops are automatically prevented.

You'll probably have noticed that none of the above mentions HTTP requests. That's because srap doesn't care - it has no idea what HTTP even is! All of the actual scraping *logic* is completely defined by the configuration - and that's what makes it a codebase. This is the scraping logic for extracting products from an LCSC category, for example. This is also why each page is its own item; that allows srap to rate-limit requests despite having absolutely no hooks into the HTTP library being used, by virtue of limiting each task to 1 HTTP request.

There are more features in srap, like deliberately invalidating past scraping results, item merges, and 'out of band' task result storage, but these are the basic concepts that make the whole thing work. As you can see, it's highly flexible, unopinionated, and easy to collaboratively maintain a scraper configuration for - every task functions more or less independently.

The datasheet search frontend

If you've used the datasheet search, you've probably noticed that it's *really* fast, it almost feels like it's all local. But no, your search queries really *are* going to a server. So how can it be that fast?

It turns out to be surprisingly simple: by default, the search is a *prefix search* only. That means that it will only search for items that *start with* the query you entered. This is usually what you want when you search for part numbers, and it *also* has some very interesting performance implications - because a prefix search can be done entirely on an index!

There's actually very little magic here - the PostgreSQL database that runs behind the frontend simply has a (normalized) index on the column for the part number, and the server is doing a `LIKE 'yourquery%'` query against it. That's it! This generally yields a search result in under 2 milliseconds, ie. nearly instantly. All it has to do is an index lookup, and those are *fast*.

On the browser side, things aren't much more complicated. Every time the query changes, it makes a new search request to the server, cancelling the old one if one was still in progress. When it gets results, it renders them on the screen. That's it. There are no trackers on the site, no weird custom input boxes, nothing else to slow it down. The result is a search that feels local :)

The source code

Right now, the source code for all of these things lives across three repositories:

- joepie91/srap - the scraping server.

- seekseek/scraper-config - the configuration and scraping logic that's used for SeekSeek.
- seekseek/ui - the search frontend (including search server!) for SeekSeek.

At the time of writing, documentation is still pretty lacking across these repositories, and the code in the srap and UI repositories in particular is pretty rough! This will be improved upon quite soon, as SeekSeek becomes more polished.

Final words

Of course, there are many more details that I haven't covered in this post, but hopefully this gives you an idea of how SeekSeek is put together, and why!

Has this post made you interested in working on SeekSeek, or maybe your own custom srap-based project? Drop by in the chat! We'd be happy to give you pointers :)

Revision #1
Created 2024-12-10 23:15:57 UTC by joepie91
Updated 2024-12-10 23:19:37 UTC by joepie91