Aggre-Gator

An RSS & Atom Feed Reader

Gus Johannesen Chris Ross
Ellis Weaver-Kreider Yanxi Wei

CSC 468 – Introduction to Cloud Computing
Dr. Linh B. Ngo — Spring 2025
West Chester University of Pennsylvania

Overview

Many sites publish RSS or Atom feeds, including YouTube, GitHub, and many others. An RSS reader allows a user to subscribe to several of these feeds, and displays updates. We propose a web-based reader, Aggre-Gator, that syndicates an arbitrary number of RSS and Atom feeds, presents them in a user-friendly way, and syncs across devices.

Chapter 1

The core functionality of Aggre-Gator is fairly similar to other feed readers that run as local applications. However, it has some notable differences on account of being cloud-based. The basic function of an RSS/Atom reader is:

- The user configures a list of feeds they are interested in.
- The user instructs the reader to refresh the feeds.
- The reader fetches a list of articles from each feed.
- The reader presents the articles to the user, both as a link and potentially as content.
- The reader saves the articles for later viewing.

Being cloud-based adds additional advantages:

- We can refresh feeds in the background, between site visits, so the user doesn't miss articles.
- Instead of a local application tied to a specific operating system or platform, our app is accessible everywhere.
- The user's subscriptions and article library follow them between devices.

Chapter 2

Aggre-Gator is divided into several logical blocks. The frontend is a static site with client-side interactivity, and is compiled to a static asset bundle which we serve with *nginx* [16]. *nginx* also proxies the `/api` route to the `backend-api` service, which interacts with the database. The feed fetcher service will periodicly update all feeds, and write the updated content to the database.

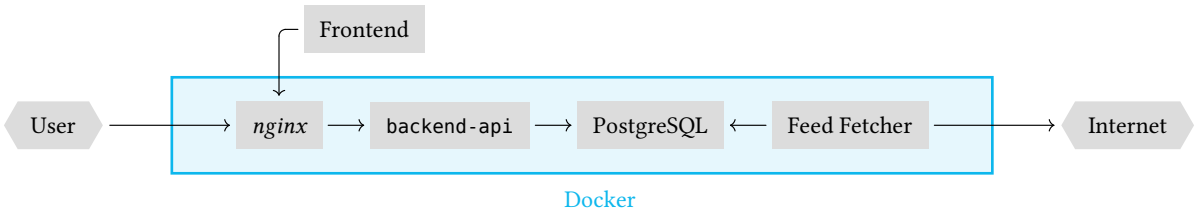

Figure 1: Design of Aggre-Gator

Frontend

Because the frontend is a static site, we have considerable flexibility deploying it. Currently, we simply serve it with *nginx*, but we have the flexibility to serve it using a CDN or with any sort of intermediate caching. In addition, we can set all assets except HTML and the favicon (and *maybe* a couple other things) to be cached indefinitely in the browser, which decreases both load times and the load on our servers.

This caching is possible because our build tool, Vite [19], gives all assets names based on the hash of their contents. Vite also bundles and minifies assets, and can be extended with plugins. We use plugins to further optimize and process images and CSS.

Instead of plain JavaScript, we opted to use Typescript [7] for better error checking and IDE integration. We also have ESLint [9] set up to catch errors and mistakes that the Typescript compiler might miss. Given that we don't need to manage any client-side UI state, we didn't use a Javascript framework. For styling, we are using Tailwind CSS [18] with WCU theme colors.

Nginx

nginx is the only publicly exposed network service in our design. This means that user-facing concerns like SSL/TLS, rate limiting, and HTTP/2 and HTTP/3 are managed in one place, rather than scattered throughout the codebase.

We have the root `/` route configured to serve assets from our frontend bundle. `/api/` is configured to proxy to the `backend-api` container. By separating our content server and backend, we are able to scale them independantly in the future. *nginx* allows us to proxy a route like `/api` to a collection of backing servers, configured in an `upstream` block [17]. We also make sure that `backend-api` is stateless, ensuring the application behaves consistently regardless of which *nginx* or `backend-api` instance a request happens to be routed through.

`backend-api`

The `backend-api` service provides a REST API to the frontend to manipulate the database, and does relatively little data processing itself. Because only the frontend is publicly exposed to the internet, the communication from *nginx* to `backend-api` and from `backend-api` to the database is unencrypted. Sending traffic on the internal container network requires access to the container runtime (Docker [1] or Podman [13]), and if an attacker has access to the runtime, they also gain the ability to read service credentials.

Database

We use PostgreSQL [10] to manage our database. We have a small binary written in Rust, [15] called `db-init`, which initializes the database cluster if it doesn't exist, and replaces itself with the Postgres server if it does exist.

Postgres has the ability to store text data such as article content in the `text` type [11], and, should we need it, can also store JSON efficiently and in a queryable manner in the `jsonb` type [12]. We believe that for this project, any document-style data is best stored in a battle-tested DBMS like Postgres, as opposed to a dedicated document-only database like Mongo [8].

In the future, we may deploy at multiple global points of presence. We may then consider a different DBMS, such as Cassandra, aimed at global-scale data replication.

Feed Fetcher

The feed fetcher will periodically get a list of feeds to refresh from the User DB, fetch the content from the web, and store the result in the Article & Feed DB. Having this as a separate component will hopefully allow us to scale it separately from other parts of the backend.

Chapter 3

To build the Docker (or more accurately OCI) containers, we use Nix [3], specifically the *nixpkgs* [2] function `pkgs.dockerTools.streamLayeredImage`. We chose this both because of the advantages to `streamLayeredImage`, and because we were already using Nix. *nixpkgs* is the largest single repository of software in the world, and one of the most up-to-date [6]. In addition, `streamLayeredImage` is reproducible [14], which makes our deployment more auditable, and less likely to break because of changes to external systems. `streamLayeredImage` also includes only the software we run and its dependencies. This means that our container images are smaller and faster to deploy. It also means that should an attacker get the ability to execute arbitrary commands, they do not have access to a shell or standard tools like

`/bin/rm` and similar (Postgres requires `/bin/sh` to run, but still does not contain things like `chmod`). Below is the code to generate an image for our `backend-api` component, taken from `flake.nix` in our repo [5].

```
backend-api-container-stream = { /*...*/ }: dockerTools.streamLayeredImage {
  name = "backend-api";
  tag = "latest";

  config = {
    Entrypoint = [
      "${lib.getExe backend-api}"
    ];
    Env = [
      "DB_HOST=db"
      "LISTEN_PORT=80"
    ];
    ExposedPorts = {
      "80/tcp" = { };
    };
  };
};
```

Figure 2: Nix code for our `backend-api` OCI container

Nix in CI Tests

We also use Nix in our CI testing. In Nix, we can create a derivation (similar to a package) and require that it build for every commit. We require that all derivations for all our components build before a PR can be merged, to ensure `main` is always in a functional state. We can also create derivations that do not produce meaningful output, but simply check something. For instance, we have a check called `formatting` that ensures that all code in the repo is formatted according to a list of formatters. We also use Nix to build this report, using Typst [4]. We have a check to ensure that before a PR that changes the report source can be merged, the `report.pdf` file must be updated to match.

Building our software and running our tests in Nix gives us some significant benefits. Because Nix carefully tracks all inputs to a derivation, it can avoid rebuilding/retesting when it can be reasonably certain the outcome will be the same. If we try to build something using the exact same source code, dependencies, and environment variables as a previous build, Nix will simply reuse that result. We can achieve a somewhat similar result with more conventional CI systems like GitHub actions, but it requires manually configuring what paths trigger rebuilds, and that these configured paths be kept perfectly up to date.

Using Nix Flakes allows us to pin all our dependencies, including all compilers and even `glibc`, to ensure our build and test environment doesn't change without us changing it deliberately. This pinned environment is the same between development, testing, and deployment. This means that in dev, we can run the exact checks that would run in CI with `nix flake check`, before commiting or pushing.

Chapter 4

Assessing the progress our team has made in developing Aggre-Gator, we have successfully implemented most of the technical requirements planned in Chapter 2, though some components are still in development.

Frontend Development

As outlined in Chapter 2, we built a static site with client-side interactivity using TypeScript. We've implemented all the core pages needed for an RSS reader, including:

- A home page that displays articles from all subscribed feeds
- A feed management page that allows users to add, view, and delete feeds
- An article detail page that displays the content of a selected article
- A bookmarks page for saved articles

Backend Development

The backend components are still in active development. While we have designed the REST API specification for communication between the frontend and database, the actual implementation of the backend-api service is still in progress. Our API specification includes endpoints for:

- Managing user feed subscriptions
- Retrieving feed information and articles
- Handling bookmarks and read/unread states

The feed fetcher service, which will periodically update subscribed feeds, is also under development. We've completed the design phase but have yet to fully implement the service.

References

[1] Docker, Inc. Docker: Accelerated Container Application Development. Retrieved from https://www.docker.com/

[2] Eelco Dolstra and Nixpkgs/NixOS contributors. NixOS/nixpkgs. Retrieved from https://github.com/NixOS/nixpkgs

[3] Eelco Dolstra. 2006. The purely functional software deployment model. Doctoral dissertation. Utrecht University. Retrieved from https://edolstra.github.io/pubs/phd-thesis.pdf

[4] Martin Haug and Laurenz Mädje. Typst. Retrieved from https://typst.app/

[5] Gus Johannesen, Chris Ross, Ellis Weaver-Kreider, Yanxi Wei, and Ellis Weaver-Kreider. Aggregator Source Code Repository. Retrieved from https://github.com/Spoonbaker/csc468-project

[6] Dmitry Marakasov. Repology - Repositories Graphs. Retrieved from https://repology.org/repositories/graphs

[7] Microsoft and TypeScript contributors. TypeScript: JavaScript with Syntax for Types. Retrieved from https://www.typescriptlang.org/

[8] MongoDB, Inc. MongoDB. Retrieved from https://www.mongodb.com/

[9] OpenJS Foundation and ESLint contributors. ESLint: Find and fix problems in your JavaScript code. Retrieved from https://eslint.org/

[10] PostgreSQL Global Development Group. PostgreSQL Database Management System. Retrieved from https://www.postgresql.org/

[11] PostgreSQL Global Development Group. PostgreSQL Documentation: Character Types. Retrieved from https://www.postgresql.org/docs/current/datatype-character.html

[12] PostgreSQL Global Development Group. PostgreSQL Documentation: JSON Types. Retrieved from https://www.postgresql.org/docs/current/datatype-json.html

[13] Red Hat, Inc. and Podman contributors. Podman. Retrieved from https://podman.io/

[14] Reproducible Builds. Reproducible Builds Website. Retrieved from https://reproducible-builds.org/

[15] The Rust Project Contributors. Rust: Empowering everyone to build reliable and efficient software. Retrieved from https://www.rust-lang.org/

[16] Igor Sysoev and Nginx, Inc. Nginx. Retrieved from https://nginx.org/en/

[17] Igor Sysoev and Nginx, Inc. Nginx Documentation: Module ngx_http_upstream_module. Retrieved from https://nginx.org/en/docs/http/ngx_http_upstream_module.html#upstream

[18] Tailwind Labs, Inc. Tailwind CSS: Rapidly build modern websites without ever leaving your HTML. Retrieved from https://tailwindcss.com/

[19] ZoidZero, Inc. and Vite contributors. Vite: Next Generation Frontend Tooling. Retrieved from https://vite.dev/