RSS Server Side Reader

Jun 26, 2025

I like the idea of RSS, but none of the RSS readers stuck with me, until I implemented one of my own, using a somewhat unusual technique. There's at least one other person using this approach now, so let's write this down.

About RSS

Let me start with a quick rundown of RSS, as the topic can be somewhat confusing. I am by no means an expert; my perspective is amateur.

The purpose of RSS is to allow blog authors to inform the readers when a new post comes out. It is, first and foremost, a notification mechanism. The way it works is that a blog publishes a machine readable list of recent blog entries. An RSS reader fetches this list, persists the list in the local state, and periodically polls the original site for changes. If a new article is published, the reader notices it on the next poll and notifies the user.

RSS is an alternative to Twitter- and HackerNews-likes for discovering interesting articles. Rather than relying on word of mouth popularity or algorithmic feeds, you curate your own list of favorite authors, and use a feed reader in lieu of compulsively checking personal websites for updates directly.

There are several specific standards that implement the general feed idea. The original one is RSS. It is a bad, ambiguous, and overly complicated standard. Don't use it. Instead, use Atom, a much clearer and simpler standard. Whenever "RSS" is discussed (as in the present article), Atom is usually implied. See Atom vs. RSS for a more specific discussion on the differences.

While simpler, Atom is not simple. A big source of accidental complexity is that Atom feed is an XML document that needs to embed HTML. HTML being almost like XML, it is easy to mess up escaping. The actually good, simple standard is JSON Feed.

However, it appears to be completely unmaintained as of 2025. This is very unfortunate. I hope someone takes over the maintenance from the original creators.

About Feed Readers

As I've mentioned, while I like the ideas behind RSS, none of the existing RSS readers worked for me. They try to do more than I need. A classical RSS reader fetches full content of the articles, saves it for offline reading and renders the content using an embedded web-browser. I don't need this. I prefer reading the articles on the author's website, using my normal browser (and, occasionally, its reader mode). The only thing I need is notifications.

What Didn't Work: Client Side Reader

My first attempt at my own RSS reader was to create a web page that stored the state in the browser's local storage. This idea was foiled by CORS. In general, if a client-side JavaScript does a `fetch` it can only fetch resources from the domain the page itself is hosted on. But feeds are hosted on other domains.

What Did Work: SSR

I have a blog. You are reading it. I now build my personalized feed as a part of this blog's build process. It is hosted at

https://matklad.github.io/blogroll.html

This list is stateless: for each feed I follow, I display the latest three posts, newer posts on top. I don't maintain read/unread state. If I don't remember whether I read the article or not, I might as well re-read! I can access this list from any device.

While it is primarily for me, the list is publicly available, and might be interesting for some readers of my blog. Hopefully, it also helps to page-rank the blogs I follow!

The source of truth is the blogroll.txt. It is a simple list of links, with one link per line. Originally, I tried using OPML, but it is far too complicated for what I need here, and is actively inconvenient to modify by hand.

Here's the entire code to fetch the blogroll, using this library:

```ts
// deno-lint-ignore-file no-explicit-any
import { parseFeed } from "@rss";

export interface FeedEntry {
  title: string;
  url: string;
  date: Date;
}

export async function blogroll(): Promise<FeedEntry[]> {
  const urls =
    (await Deno.readTextFile("content/blogroll.txt"))
    .split("\n").filter((line) => line.trim().length > 0);
  const all_entries =
    (await Promise.all(urls.map(blogroll_feed))).flat();
  all_entries.sort((a, b) =>
    b.date.getTime() - a.date.getTime());
  return all_entries;
}

async function blogroll_feed(
  url: string
): Promise<FeedEntry[]> {
  let feed;
  try {
    const response = await fetch(url);
    const xml = await response.text();
    feed = await parseFeed(xml);
  } catch (error) {
    console.error({ url, error });
    return [];
  }

  return feed.entries.map((entry: any) => {
    return {
      title: entry.title!.value!,
      url: (entry.links.find((it: any) => {
        it.type == "text/html" || it.href!.endsWith(".html");
      }) ?? entry.links[0])!.href!,
      date: (entry.published ?? entry.updated)!,
    };
  }).slice(0, 3);
}
```

And this is how the data is converted to HTML during build process:

```ts
export function BlogRoll(
  { posts }: { posts: FeedEntry[] }
) {
  function domain(url: string): string {
    return new URL(url).host;
  }

  const list_items = posts.map((post) => (
```

```
    <li>
      <h2>
        <span class="meta">
          <Time date={post.date} />, {domain(post.url)}
        </span>
        <a href={post.url}>{post.title}</a>
      </h2>
    </li>
  ));

  return (
    <Base>
      <ul class="post-list">
        {list_items}
      </ul>
    </Base>
  );
}
```

GitHub actions re-builds blogroll every midnight:

```
name: CI
on:
  push:
    branches:
      - master

  schedule:
    - cron: "0 0 * * *" # Daily at midnight.

jobs:
  CI:
    runs-on: ubuntu-latest
    permissions:
      pages: write
      id-token: write

    steps:
      - uses: actions/checkout@v4
      - uses: denoland/setup-deno@v2
        with:
          deno-version: v2.x

      - run: deno task build --blogroll

      - uses: actions/upload-pages-artifact@v3
        with:
          path: ./out/www
      - uses: actions/deploy-pages@v4
```

Links

Tangentially related, another pattern is to maintain a list of all-times favorite links:

https://matklad.github.io/links.html