Vepsäläinen, Juho; Vuorimaa, Petri

Bridging Static Site Generation with the Dynamic Web

Published in:
Web Engineering - 22nd International Conference, ICWE 2022, Proceedings

DOI:
10.1007/978-3-031-09917-5_32

Published: 01/01/2022

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
Vepsäläinen, J., & Vuorimaa, P. (2022). Bridging Static Site Generation with the Dynamic Web. In T. Di Noia, I.-Y. Ko, M. Schedl, & C. Ardito (Eds.), *Web Engineering - 22nd International Conference, ICWE 2022, Proceedings* (pp. 437-442). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 13362 LNCS). Springer. https://doi.org/10.1007/978-3-031-09917-5_32

Bridging static site generation with the dynamic web*

Juho Vepsäläinen[1] and Petri Vuorimaa[1]

Aalto University
https://www.aalto.fi/en/department-of-computer-science
petri.vuorimaa@aalto.fi

Abstract. Historically web sites have been developed using HTML for their markup either by authoring it directly or through abstraction to generate it. The currently available tools exist in a continuum of static, developer-oriented tools and dynamic services that cater to non-technical users. In this paper, we propose an approach that sits in the middle by using JSON for site definitions. The definition is leveraged on the client-side for editing, bridging the continuum's ends.

Keywords: Static site generation · JSON · Software architecture

1 Brief introduction to static site generation

Static Site Generators (SSG) have become a hot topic in the developer community as they provide a set of benefits over more traditional server-based dynamic approaches:

- Fast page load time: As there's less processing to do, also pages tend to load faster [1] [2]
- Efficient scalability: By definition, it's easy to scale static sites across multiple servers on a CDN [1]
- Availability and security: Since there's no dynamic server portion, also attack surface is largely reduced and the server logic required is a magnitude simpler in terms of logic [1] [2]
- Versioning: As static sites are versioned automatically, it is easy to track changes using systems such as Git or GitHub [2]

As a side effect, static sites consume fewer resources and therefore consume fewer natural resources than their dynamic brethren [1].

* Supported by Aalto University.

1.1 Challenges of SSG

The main drawback of SSG has to do with the need to **write** and to alter data as that's the sweet spot for dynamic sites. In the worst case for SSG, an update to the content will force you to regenerate and deploy the entire site. In a dynamic solution generating a page using a database, the problem is close to non-existent.

The main challenge for SSG is to retain the benefits while addressing the cost of updates and allowing easy edits for the editors of a site. It is good to note that although the developer and the site's editor can be the same person, it's not always so, and that's when different expectations for User Experience (UX) might arise. Something suitable for developers isn't necessarily good for editors and vice versa.

1.2 SSG with a headless content server

When site content is coupled with its markup, the site editors have to worry about both when editing. For this reason, so-called headless content servers have appeared on the market. [3]

They are a step forward, but they still require developer attention and have limitations in terms of flexibility. Anything that should be modifiable has to show up in the data model.

2 SSG using JSON as an intermediate target

In this paper, we propose using JSON as an intermediate target to define aspects, such as site markup, styling, state management, content binding, and routing. As a side effect of doing this, developing an editor that runs on top of the deployed site itself becomes possible.

2.1 Why JSON?

JSON is the universal data format of the web, and handling is built right into JavaScript. You could replace it with any other format, but it has proven to be a good fit for demonstrating the ideas due to its relative simplicity. According to Nurseitov et al. [4], JSON is more performant than its XML counterpart, making the choice even more lucrative.

2.2 Constraints

For the approach to work, it is paramount that the definitions capture it all. Due to this constraint, commonly used techniques, such as using separate CSS files for styling, become restricted and are limited only for special cases such as highlighting code. The same goes for aspects like state management, as ideally, they should be handled within the HTML markup.

2.3 Aspects of the architecture

Given the constraints, several technical decisions must be made for the system to exist. I've listed my choices and related options below[1]:

- Routing: custom JSON definition to describe data sources, possible data transforms, routes, route expansions, metadata, which layouts to use for rendering pages, rendering target (HTML, XML)
- Layouting and components: custom JSON definition that's mapped to HTML or XML depending on the target
- Data binding: custom `__bind` property to bind data to the current scope
- State management: Sidewind[2], a state manager that comes with a small runtime and allows state management within HTML
- Styling: Twind, a derivative of Tailwind, a popular utility CSS solution that allows authoring styling within HTML

2.4 Routing

Each site has a routing scheme that defines the available pages. In dynamic sites, the routes can be created on-demand, but in SSG they need to be predefined on some level, although mixing the approaches is possible. In other words, dynamic functionality can be overlaid on top of a static site, but that's another discussion.

For a static site, page structure can be either defined implicitly through file structure and a convention or configuration. In the former approach, routing information is encoded to file naming and the way they are laid out.

In the latter approach, a mapping between routes and files is defined using a separate configuration file. After realizing it gives an additional degree of flexibility while being explicit and declarative by nature, we chose this approach.

A good routing approach should fulfill at least the following criteria:

1. It should be easy to understand the existing routes and add new ones
2. It should be possible to define how routes are connected to data and which layout is used for rendering them
3. It should be possible to attach metadata per route for Search Engine Optimization (SEO) purposes
4. It should be possible to map arbitrary input data to a page or multiple pages, assuming it's a collection

To fulfill the first three criteria, we ended up with the following JSON definition where the object key defines the route name, and then within the value properties, the route is declared in detail.

[1] The selection has been explored in a personal project called Gustwind (https://gustwind.js.org/)
[2] https://sidewind.js.org/

In the example below, project README.md file is mapped to the site's index while pushing it through a Markdown transform that's converting it to HTML. Then it's rendered using a layout, and the layout also has access to meta-information related to it.

```
{
  "/": {
    "meta": { "title": "Gustwind", ... },
    "layout": "siteIndex",
    "dataSources": [
      {
        "id": "readme",
        "operation": "file",
        "input": "./README.md",
        "transformWith": ["markdown"]
      }
    ]
  }
}
```

The root route ("/") of the site is a particular case, and for any other route, a simple name can be used (i.e., "blog"). The routes can also be nested arbitrarily by using a "routes" field inside which you can use the same definition.

Handling the fourth criteria of mapping data to multiple pages is more complex, and we ended up implementing expansion syntax for this purpose as follows:

```
{
  "blog": {
    ...
    "expand": {
      "dataSources": [ { ... } ],
      "matchBy": {
        "dataSource": "blogPosts",
        "collection": "content",
        "slug": "data.slug"
      },
      "layout": "blogPage",
      "meta": { "__title": "match.name", ... }
    }
  }
}
```

The trick was to allow matching against processed data to generate pages. The use case is typical, especially when you have a group of files you wish to map to a website, so it made sense to support it.

2.5 Components

Components are a useful abstraction in web development as they allow you to capture meaningful entities for reuse. They allow you to capture repetition within a single interface, and the advent of front-end frameworks and Web Components[5] has made them a popular approach.

We've ended up with the following component definition:

```
{
  "element": "nav",
  "class": "sticky top-0",
  "children": [ ... ]
  "attributes": {
    "title": "Navigation"
  }
}
```

Above would map as HTML like this:

```
<nav class="sticky top-0" title="Navigation">...</nav>
```

2.6 Layouts and data binding

For layouts, we consider the following criteria important:

1. It should be possible to connect to data from a layout
2. It should be possible to compose layouts and components for reuse
3. It should be possible to perform nested data binding to allow passing specific data to components

The following example illustrates a whole layout with head and body sections inside which components are then bound to data using the `__bind` property. In addition, `__` prefix is used to bind data to `children` for iteration, and the same convention can be used for binding to attributes as well:

```
{
  "head": [ { "element": "MetaFields" } ],
  "body": [
    { "element": "MainNavigation" },
    {
      "element": "div",
      "class": "md:mx-auto my-8 px-4 md:px-0 w-full",
      "__bind": "readme",
      "__children": "content"
    },
    { "element": "MainFooter" }
  ]
}
```

To capture a subset of data for a component, `__bind` can be applied anywhere within the component tree. The scheme could be expanded to support React.js style props and concrete component examples to document it.

3 Discussion

Due to space constraints, we didn't cover styling and state management in detail. However, they fit the same scheme well and allow development of full-fledged web sites and applications. We believe the approach yields the following benefits that are to be proven and give a starting point for further research:

1. Given the system uses JSON as an intermediate format, is it possible to develop client-side editors on top of it by providing a JSON definition next to an HTML page rendered by the system?
2. Given it's possible to mix the generated pages with dynamic content by using upcoming web technologies, such as edge computing, can the approach be used to mix the techniques?
3. The usage of JSON yields further value in knowledge sharing. Could, for example component registries be implemented on top of the approach?
4. Does the architecture scale to using legacy technologies within it, for example using the islands or widget architectures?
5. Which different techniques can be leveraged on top of the approach to improve it further? What benefits do they provide?
6. How does the approach compare with established SSGs?

Completing the work would have the potential to provide a new approach for developing websites and applications. Due to the nature of the specification, it could become a standard for different technical implementations besides the reference one allowing usage across multiple platforms, and the ideas might scale beyond the web.

References

1. Petersen, Hillar. "From Static and Dynamic Websites to Static Site Generators." University of TARTU, Institute of Computer Science (2016).
2. Camden, Raymond, and Brian Rinaldi. Working with Static Sites: Bringing the Power of Simplicity to Modern Sites. " O'Reilly Media, Inc.", (2017).
3. Öfverstedt, Linn. "Why go headless–a comperative study between traditional CMS and the emerging headless trend." Uppsala University, Department of Information Technology (2018).
4. Nurseitov, Nurzhan, et al. (2009). Comparison of JSON and XML data interchange formats: A case study. 22nd International Conference on Computer Applications in Industry and Engineering 2009, CAINE 2009. 157-162.
5. Pelkonen, Hannu. "Web components as application building blocks." Aalto University. School of Science. MS thesis. (2014).