Creating technology glossary

As part of creating a writing guide for my content I've been thinking about how to create a glossary of technical terms that we can expand as we go along.

Yahoo! Style Guide provided such a glossary and made it available for download so people could use it as is or use it as a starting point for their own glossaries.

The project tries to create a glossary of relevant terms for a technology blog. Some steps leading to this are:

1. Convert the Yahoo Style Guide to JSON to use as the base for the glossary
2. Decide if this is single page or multi-page application
3. Evaluate Mavo to create the glossary pages

Converting XML to JSON

The first step is to take the Yahoo! Glossary document and convert it to JSON. I used oXygen's XML to JSON converter to get the JSON file.

I validated the JSON file using JSONLint and it passed.

We now have a JSON file ready to use. Let's look at the other side of the project, the UI and the tools to create it.

Single Page Application?

Before we decide on the tools to create the glossary we need to decide if this is going to be a Single Page Application or and old school website with multiple pages, one per letter with content.

I think it's best if we start with a single page application containing all the glossary content. We can then decide if that's the best structure moving forward.

Another look at Mavo

Mavo shows an interesting way to build web content. It's not a framework in the React/Angular/Vue sense but it provides all the basic blocks to build web applications along with extensibility to add new components and enhance existing ones using Javascript.

All the basic application setup is done in markup with attributes on standard HTML elements. Mavo will read and process these attributes to build the application.

Mavo also has a plugin ecosystem to enhance the system's functionality. There are three plugins that I'm interested in evaluating:

- Mavo-Markdown - Adds Markdown support
- Mavo Cropper - Adds image cropping and upload
- Mavo Sort - provides sorting

Building a proof of concept

Building a Mavo application is not hard. Most of the work is done in markup. I only have to set up the data storage and the plugins.

This is the complete markup for the first version of the glossary application.

```
<head>
  <title>T<title>l Glossary</title>
  <meta charset="</title>    <meta name="viewport" content="width=device
  <<meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="styles/main.css">
  <script src="https://get.mavo.io/mavo.js"><link rel="stylesheet" href=
  <script src="scripts/mavo-cropper.js"></script>
  <script src="scripts/mavo-sort.js"></script>

</head>
<body mv-app="Glossary"
      mv-storage="https://github.com/caraya/mglossary/blob/main/glossary
      mv-plugins="mavo-markdown mavo-cropper">
```

```
    <h1>Technical Glossary</h1>

    <div class="Glossary" mv-list mv-sort="+">

      <!--
        The section and its content </script>     <!--
        The section and its content will be repeated for each entry in the
      -->
      <section  typeof="glossary-entry"
        mv-list-item
        class="glossary"
        mv-plugins="markdown mavo-cropper"
        mv-sort="+">
```

```
        <h2 property="Word" mv-default="Default Glossary Entry"></h2>
```

```
        <fieldset>
          <legend>Description</legend>
          <div property="Description" mv-default="Description" class="mark
        </fieldset>
```

```
        <fieldset>
          <legend>Associated Image</legend>
          <img property="Image" mv-default="" class="glossary-entry-image"
        </fieldset>
      </section>
    </div>
```

The scripts and styles load the code for Mavo to run.

The attributes in the body element and its children tell Mavo what do with the content.

We will dissect how the application works in more detail in the following sections.

Building the application

Loading Mavo and plugins

Mavo loads like any other script and, as such, can be loaded locally or from a CDN. The glossary uses both approaches.

We load the mavo application script and style sheet from a CDN to make sure they remain current.

We load the scripts and styles for the plugins from local copies. As far as plugins go, I want to ensure they work as intended and don't change on me.

The code below loads the Mavo application and the plugins from the head of the page they are hosted on.

```
<link rel="stylesheet" href="https://get.mavo.io/mavo.css"/>
<link rel="stylesheet" href="styles/mavo-markdown.css">
<lin<link rel="stylesheet" href="styles/mavo-markdown.css">"https://get.ma
<script src="scripts/mavo-markdown.js"></script>
<script src="scripts/mavo-cropper.js"></script><script src="scripts/mavo-
<script src="scripts/mavo-sort.js"></script>
```

Initializing Mavo

Initializing Mavo is relatively easy. We use the `mv-app` attribute to indicate the name of our application. The name must be unique in the page since Mavo allows more than one application per page.

`mv-storage` tells Mavo where to store the data. In this case we use Github and, for that to work, we need to set up a repository for the application and provide the full URL (with an optional file name) as the attribute value.

Mavo will only allow people with a Github account and write access to the repository, assigned by the repository's owner, to manipulate data for the application. For everyone else, the application is read-only.

The `mv-plugins` attribute tells the application what plugins the application will use. The application will load the `mavo-markdown` and `mavo-cropper` plugins.

```
<body mv-app="Glossary"
  mv-storage="https://github.com/caraya/mglossary/blob/main/glossary.json"
  mv-plugins="mavo-markdown mavo-cropper">
```

Once this is done we can start writing the markup for the application.

Writing the application markup

The core of the application is the div with the class `Glossary` and the children inside. This is the container for the list of glossary entries.

The main container get a class assignment and two additional attributes: `mv-list` tells Mavo that the children inside are list items and `mv-sort="+"` tells mavo to sort the children.

```
<div  class="Glossary"
      mv-list
      mv-sort="+">
```

The wrapper for each entry is a `section` element. We qualify the section as a glossary entry by adding the `typeof` attribute.

We tell it that this is a repeatable element using the `mv-list-item` attribute.

We ensure that the individual list item is sortable using `mv-sort="+"`.

```
<section  typeof="glossary-entry"
  mv-list-item
  class="glossary"
  mv-sort="+">
```

Mavo will take all the elements with a `porperty` attribute and make them editable.

It is also a good practice to provide a `mv-default` attribute for each element.

```
<h2 property="Word"
```

```
    mv-default="Default Entry"></h2>
```

The description element uses the `property` and `mv-default` attributes.

It also signals Mavo that the content can take Markdown syntax by adding the `class="markdown"` attribute.

```
<fieldset>
  <legend>Description</legend>
  <div property="Description"
  mv-default="Description"
  class="markdown"></div>
</fieldset>
```

The associated image area is similar to the description area but uses an image element instead of a div. We also use the `property` and `mv-default` attributes.

However because it uses an `img` element, Mavo will treat it differently. Without any additional code, Mavo will provide an upload button and the `mavo-cropper` plugin will provide tools to crop the image before uploading it.

```
<fieldset>
  <legend>Associated Image</legend>
  <img  property="Image"
    mv-default=""
    class="glossary-entry-image">
</fieldset>

</section>
</div>
```

The code portion of the application is now complete. We'll look at some possible enhancements when we talk about possible future changes.

Basic styling

The styles for the application are also simple. It's all contained in the `main.css`

file.

The main font is Recursive, a variable font that produces a variety of different styles from the same basic font file.

Whenever working with a new font, I run it through Wakamaifondue to generate CSS for all the open type features available to the font, as well as all custom defined combination of variable features.

These features are not relevant to the basic styles of the application so they won't be mentioned in this section.

Because this is a proof of concept it is acceptable to only use WOFF2 fonts. For a production site, WOFF and WOFF2 fonts are necessary.

```
@font-face {
  font-family: 'Recursive'Recursive'url('../fonts/Recursive_VF_1.078.woff2
  font-weight: 300 1000;
  font-display: swap;
}

b'woff2'bodyt-family: "Recursive", sans-serif;
  line-height: 1.35;
}

.Glossar"Recursive".Glossary {
  margin: 0 auto;
  width: 60vw;
}

h2 {
  color: rebeccapurple;
  font-weight: 700;
}

section {
  margin-bottom: 4em;
}
```

```
div {
  padding: 2rem;
  height: auto;
}

img {
  max-width: 50%;
  margin: 0 auto;
}
```

Future research areas

The project works but there are some areas that I would like to improve upon.

Github as a storage solution works but it may be problematic for large data sets. Moving storage to Firebase's Firestore database may provide a better solution.

There is a Firebase Firestore plugin available. The plugin itself is good but the documentation seems to be a full guide to using Firestore rather than how to use the Mavo plugin.

There is also an issue with data migration. How do we move the data from Github to Firestore?

Another future capability to research is how to provide offline capabilities using the mavo-offline plugin.

According to the plugin's documentation:

> Mavo-offline stores Mavo's fetched and stored data in localStorage. This means that the next time you visit the site, it will immediately show data from localStorage, and then update with the server's data when received.
>
> This also means that you can store data even when offline. It will then send it to the server when coming online again. Even if you refresh or revisit the page later.

> Mavo-offline also supports Mavo backends with server side pushes (e.g. mavo-couchdb and mavo-firebase). That means it can update the view when there have been server side changes.

If I understand the documentation correctly, when using `mavo-offline` and `mavo-firebase` you can see changes to the server in real time, regardless of where they were made.

It's an interesting idea worth further exploration.

Conclusion

Mavo provides a simple way to create web applications. It's meant to work alongside static pages and other content that doesn't require dynamic content. And for its intender purpose it shines.

This post has covered building a basic Mavo application to display data for a glossary of technical terms.

The application code is hosted on Github and it's running on Github pages at https://caraya.github.io/glossary/