Templating the web

There are times when writing the same thing oveer and over again gets to be really tedious. Think of populating large bulleted lists, select statements and other repetitive elements.

We've been able to do this for a while using libraries... We'll look at it handlebars.js and then we'll look at using the `template` tag built into the HTML specification.

We'll also look at why we might want to keep working with libraries to support older browsers and browsers with incomplete support.

The current way: Template Libraries

When I first looked at Handlebars it appeared to be too much work for the result I was looking for. As I started working with WordPress, particularly when they released their REST API, I ralized that templating would be key to build custom interfaces.

The most basic example shows how to use Handlebars to populate a template with information stored in the same script. The HTML contains both the placeholder element where we'll store the content and the script containing the actual template.

```
<div id="entries"></div>

<script id</div>y-template" type="text/x-handlebars-template">
  <div class="entry">
    <h1>{{title}}</h1>
    <div class="body">
      {{body}}
    </div>
  </div>
</script>
```

The script element captures the content of the templae in the `source` variable and defines a shorthand for template compilation on `renderEntry`.

We then define the content we want to insert. In this case it's a variable holding an array of data we want to populate the content with.

Finally we use `renderEntry` to rendeer the template, populated with the data, into HTML and insert it into our placeholder element.

```
let source = document.getElementById('entry-template').innerHTML;
let renderEntry = Handlebars.compile(source);

let blogEntry = {
  title: 'My New Post',
  body: 'This is my first post!'
};

document.getElementById('entries').innerHTML = renderEntry(blogEntry);
```

A next iteration could be used to pull from an array of values stored locally. For this we use a helper method, #each to loop through the values on the array and provide the data to populate the template.

The HTML is almost identical except that we wrap the content we'll iterate over with an #each helper (`{{#each cats}}` and `{{/each}}`. cats is a reference to the data we will pass on Javascript.

```
<div id="cat-list"></div>

<template id="cat-list-template">
  {{#each cats}}
    <div class="cat">
      <h1>{{name}}</h1>
      <p>Age: {{age}}</p>
    </div>
  {{/each}}
</template>
```

The Javascript is also similar to the prior example. The differences are:

- The data we're passing is now an array

- When we instantiate the template we wrap the data we pass in an array to make sure the helper works as intended

```
var myCats = [
  { name: 'Fiona', age: 4 },
  { name: 'Spot', age: 12 },
  { name: 'Chestnut', age: 4 },
  { name: 'Frisky', age: false },
  { name: 'Biscuit', age: 4 }
];

var template = document.getElementById('cat-list-template').innerHTML;
var renderCats = Handlebars.compile(template);
document.getElementById('cat-list').innerHTML = renderCats({
  cats: myCats
});
```

The last bit of Handlebars magic we'll look at is how to use it to render templates with external data from a REST API; in this case Wordpress.

The template is pretty similar to the cat example. The main difference is the use of nested values (`title.rendered` and `content.rendered`) and the use of the triple mustache around `content.rendered` to tell Handlebars that we don't want to escape HTML values for this variable.

If you don't own the content then please **don't do this!** In this case, since it's my blog and I'm pretty sure I don't write malware (bad code, maybe but definitely not malware) I've accepted the risk.

```
<div id="myContent"></div>

  <template id="post-list-template">

    {{#each posts}}
    <div class="post">
      <h1>{{title.rendered}}</h1>
      <div>
        {{{content.rendered}}}
```

```
      </div>
    </div>
    {{/each}}

  </template>
```

The Javascript is different. I've chosen to use fetch and promises to make the code look nicer. We could go with async and await but that would limit the code to newer browsers (yes, I know they are evergreen but I also know of IT departments that block updates or choose to use LTS/ESR versions that lag behind in features), even more so than promises do.

The code starts with a fetch request to the Wordpress REST API requesting the 4 more recent posts of my blog. You can change the value by changing the value of the `per_page` parameter. This will generate a promise that **resolves** when the fetch request completes and the data download is finished and **rejects** otherwise.

Once the promise resolves we move to the next step and convert it to JSON data using the response object's json method.

Once the promise of `response.json()` fulfills we move to the next, and final, step. We compile the template and render it using the data we just fetched. These are the same command that were at the bottom of the cat example. Since we are working with promises we must move them into the promise chain; otherwise the fetch request will complete before we reach the part of the script where we compile and render the template.

If any of the promises rejects the code will jump to the catch statement. In this case we're only logging the error to console. we might also want to display something to the user to indicate the failure. No blank pages, please.

```
let myPosts = fetch(
  'https://publishing-project.rivendellweb.net/wp-json/wp/v2/posts?per_pag
)
  .then(response => {
    return response.json();
  })
  .then(myJson => {
```

```
    let template = document.getElementById('post-list-template').innerHTML
    let renderPosts = Handlebars.compile(template);
    document.getElementById('myContent').innerHTML = renderPosts({
      posts: myJson
    });
  })
  .catch(err => {
    console.log("There's been an error getting the data", err);
  });
```

This was meant as a proof of concept and is in now way, shape or form production code. Some areas of further work:

- Caching the fetch results to improve load times after first visit
- Pagination

The template element

Rather than use a library, wouldn't it be nice if we could use native HTML to create templates and then insantiate them with Javaascript without having to add libraries and additional HTTP request.

We can!

HTML templates were first proposed as part of the web components family of specifications. They have since moved to the HTML Specification itself.

The idea behind the template element is that it holds content on the page without the content being active... we can activate the template at any time using Javascript.

Again it's worth repeating some of the characteristics of natie HTML templates:

1. Its content is effectively inert until activated. Essentially, your markup is hidden DOM and does not render. This means that script won't run, images won't load, audio won't play until the template is used
2. Content is considered not to be in the document. Using document.getElementById() or querySelector() in the main page won't

return child nodes of a template

3. Templates can be placed anywhere inside of <head>, <body>, or `<frameset>` and can contain any type of content which is allowed in those elements
    1. Note that “anywhere” means that `<template>` can safely be used in places that the HTML parser disallows…all but content model children. It can also be placed as a child of `<table>` or `<select>`

The following template can be anywhere on the page but, for consistency sake, let’s put it at the bottom of the page. Notice that we’re ok with having an empty `src` attribute for the image element

```
<template id="mytemplate">
  <img src="" alt="great image">
  <div class="comment"></div>
</template>
```

In your application script use the following script to stamp the template into the live DOM.

We first create a function to make feature detection for templates easier.

We then use the function in an if statement. Inside the if block we do the following:

- Select the template element and store the result in a variable
- Insert the path to the image inside the source attribute. We use the template’s `content` method to traverse inside the inert template
- We create a cloned compy of the template content using document.importNode
- We append the cloned node into the document.

If the user agent doesn’t support templates we can fallback to using a library like Handlebars.

```
function supportsTemplate() {
  return 'content' in document.createElement('template');
}
```

```
if (supportsTemplate()) {
  var 'template'nt.querySelector('#mytemplate');
  '#mytemplate'// Populate the src at runtime.querySelector('img').src =
  var clone = document.importNode(t.content, true);
  document.body.appendChild(clone);
} else {
  // Use old te'img'// Use old templating techniques or libraries.
}
```

The one thing I'm still working on figuring out is how you can create multiple copies of the same template and populate it with different data, like the Wordpress/Handlebars example we discussed earlier. I will update the post once I mfigure it out :)

Links and Resources

- Template Element Description at MDN
- Older Template Tutorial from HTML5 Rocks
- Handlebars.js