HTML's New Template Tag

- Feature Detection
- Declaring template content
- The pillars
- Activating a template
- Demos
  - Example: Inert script
  - Example: Creating Shadow DOM from a template
- Gotchas
  - The road to a standard
  - Method 1: Offscreen DOM
  - Method 2: Overloading script
- Conclusion
- Additional resources

The concept of templating is not new to web development. In fact, server-side templating languages/engines like Django (Python), ERB/Haml (Ruby), and Smarty (PHP) have been around for a long time. In the last couple of years however, we've seen an explosion of MVC frameworks spring up. All of them are slightly different, yet most share a common mechanic for rendering their presentational layer (aka da view): templates.

Let's face it. Templates are fantastic. Go ahead, ask around. Even its definition makes you feel warm and cozy:

template (n) - A document or file having a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used.

"...does not have to be recreated each time..." Don't know about you, but I love avoiding extra work. Why then does the web platform lack native support for something developers clearly care about?

The WhatWG HTML Templates specification is the answer. It defines a new `<template>` element which describes a standard DOM-based approach for client-side templating. Templates allow you to

declare fragments of markup which are parsed as HTML, go unused at page load, but can be instantiated later on at runtime. To quote Rafael Weinstein:

They're a place to put a big wad of HTML that you don't want the browser to mess with at all…for any reason.

Feature Detection

To feature detect `<template>`, create the DOM element and check that the `.content` property exists:

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

if (supportsTemplate()) {
    // Good to go!
} else {
    // Use old templating techniques or libraries.
}
```

Declaring template content

The HTML `<template>` element represents a template in your markup. It contains "template contents"; essentially **inert chunks of cloneable DOM**. Think of templates as pieces of scaffolding that you can use (and reuse) throughout the lifetime of your app.

To create a templated content, declare some markup and wrap it in the `<template>` element:

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

The observant reader may notice the empty image. That's perfectly fine and intentional. A broken image won't 404 or produce console errors because it won't be fetched on page load. We can dynamically generate the source URL later on.

The pillars

Wrapping content in a `<template>` gives us few important properties.

1. Its **content is effectively inert until activated**. Essentially, your markup is hidden DOM and does not render.
2. Any content within a template won't have side effects. **Script doesn't run, images don't load, audio doesn't play**,...until the template is used.
3. **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.
4. Templates **can** be placed anywhere inside of `<head>`, `<body>`, or `<frameset>` and can contain any type of content which is allowed in those elements. 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>`:

```
<table>
  <tr>
    <template id="cells-to-repeat">
      <td>some content</td>
    </template>
  </tr>
</table>
```

Activating a template

To use a template, you need to activate it. Otherwise its content will never render. The simplest way to do this is by creating a deep copy of its `.content` using `document.importNode()`. The `.content` property is a read-only `DocumentFragment` containing the guts of the template.

```
var t = document.querySelector('#mytemplate');
// Populate the src at runtime.
t.content.querySelector('img').src = 'logo.png';

var clone = document.importNode(t.content, true);
document.body.appendChild(clone);
```

After stamping out a template, its content "goes live". In this particular example, the content is cloned, the image request is made, and the final markup is rendered.

Demos

Example: Inert script

This example demonstrates the inertness of template content. The `<script>` only runs when the button is pressed, stamping out the template.

```
<button onclick="useIt()">Use me</button>
<div id="container"></div>
<script>
  function useIt() {
    var content = document.querySelector('template').content;
    // Update something in the template DOM.
    var span = content.querySelector('span');
    span.textContent = parseInt(span.textContent) + 1;
    document.querySelector('#container').appendChild(
      document.importNode(content, true)
    );
  }
</script>

<template>
  <div>Template used: <span>0</span></div>
  <script>alert('Thanks!')</script>
</template>
```

Example: Creating Shadow DOM from a template

Most people attach Shadow DOM to a host by setting a string of markup to `.innerHTML`:

```
<div id="host"></div>
<script>
  var shadow = document.querySelector('#host').createShadowRoot();
  shadow.innerHTML = '<span>Host node</span>';
</script>
```

The problem with this approach is that the more complex your Shadow DOM gets, the more string concatenation you're doing. It doesn't scale, things get messy fast, and babies start to cry. This approach is also how XSS was born in the first place! `<template>` to the rescue.

Something more sane would be to work with DOM directly by appending template content to a shadow root:

```
<template>
<style>
  :host {
    background: #f8f8f8;
    padding: 10px;
    transition: all 400ms ease-in-out;
    box-sizing: border-box;
    border-radius: 5px;
    width: 450px;
    max-width: 100%;
  }
  :host(:hover) {
    background: #ccc;
  }
  div {
    position: relative;
  }
  header {
    padding: 5px;
    border-bottom: 1px solid #aaa;
  }
  h3 {
    margin: 0 !important;
  }
  textarea {
    font-family: inherit;
    width: 100%;
    height: 100px;
    box-sizing: border-box;
    border: 1px solid #aaa;
  }
  footer {
    position: absolute;
```

```
    bottom: 10px;
    right: 5px;
  }
</style>
<div>
  <header>
    <h3>Add a Comment
  </header>
  <content select="p"></content>
  <textarea></textarea>
  <footer>
    <button>Post</button>
  </footer>
</div>
</template>

<div id="host">
  <p>Instructions go here</p>
</div>

<script>
  var shadow = document.querySelector('#host').createShadowRoot();
  shadow.appendChild(document.querySelector('template').content);
</script>
```

Gotchas

Here are a few gotchas I've come across when using `<template>` in the wild:

- If you're using modpagespeed, be careful of this bug. Templates that define inline `<style scoped>`, many be moved to the head with PageSpeed's CSS rewriting rules.
- There's no way to "prerender" a template, meaning you cannot preload assets, process JS, download initial CSS, etc. That goes for both server and client. The only time a template renders is when it goes live.
- Be careful with nested templates. They don't behave as you might expect. For example:

```
<template>
  <ul>
```

```
    <template>
      <li>Stuff</li>
    </template>
  </ul>
</template>
```

Activating the outer template will not active inner templates. That is to say, nested templates require that their children also be manually activated.

The road to a standard

Let's not forget where we came from. The road to standards-based HTML templates has been a long one. Over the years, we've come up with some pretty clever tricks for creating reusable templates. Below are two common ones that I've come across. I'm including them in this article for comparison.

Method 1: Offscreen DOM

One approach people have been using for a long time is to create "offscreen" DOM and hide it from view using the `hidden` attribute or `display:none`.

```
<div id="mytemplate" hidden>
  <img src="logo.png">
  <div class="comment"></div>
</div>
```

While this technique works, there are a number of downsides. The rundown of this technique:

- *Using DOM* - the browser knows DOM. It's good at it. We can easily clone it.
- *Nothing is rendered* - adding `hidden` prevents the block from showing.
- *Not inert* - even though our content is hidden, a network request is still made for the image.
- *Painful styling and theming* - an embedding page must prefix all of its CSS rules with `#mytemplate` in order to scope styles down to the template. This is brittle and there are no guarantees we won't encounter future naming collisions. For example, we're hosed if the embedding page already has an element with that id.

Method 2: Overloading script

Another technique is overloading `<script>` and manipulating its content as a string. John Resig was probably the first to show this back in 2008 with his Micro Templating utility. Now there are many others, including some new kids on the block like handlebars.js.

For example:

```
<script id="mytemplate" type="text/x-handlebars-template">
  <img src="logo.png">
  <div class="comment"></div>
</script>
```

The rundown of this technique:

- *Nothing is rendered* - the browser doesn't render this block because `<script>` is `display:none` by default.
- *Inert* - the browser doesn't parse the script content as JS because its type is set to something other than "text/javascript".
- *Security issues* - encourages the use of `.innerHTML`. Run-time string parsing of user-supplied data can easily lead to XSS vulnerabilities.

Conclusion

Remember when jQuery made working with DOM dead simple? The result was `querySelector()` / `querySelectorAll()` being added to the platform. Obvious win, right? A library popularized fetching DOM with CSS selectors and standards later adopted it. It doesn't always work that way, but I *love* when it does.

I think `<template>` is a similar case. It standardizes the way we do client-side templating, but more importantly, it removes the need for our 2008 hacks. Making the entire web authoring process more sane, more maintainable, and more full-featured is always a good thing in my book.

Additional resources

- WhatWG Specification
- Introduction to Web Components
- <web>components</web> (video) - a fantastically comprehensive presentation by yours truly.

□□□ #3
□ □□□ □□□ 2026-10-31 22:28:15 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:28 UTC