Using the HTML5 History API

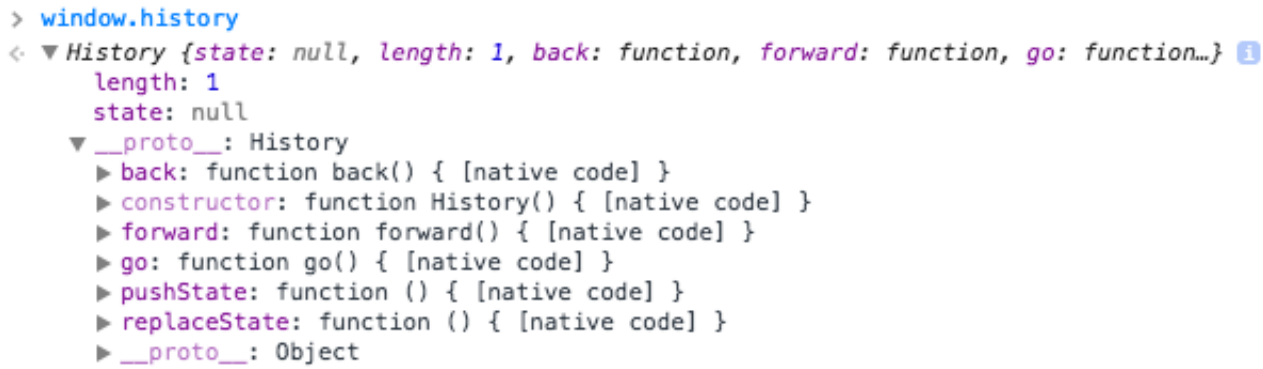
March 9, 2015

The HTML5 History API gives developers the ability to modify a website's URL without a full page refresh. This is particularly useful for loading portions of a page with JavaScript, such that the content is significantly different and warrants a new URL.

Here's an example. Let's say a person navigates from the homepage of a site to the Help page. We're loading the content of that Help page with Ajax. That user then heads off to the Products page which we again load and swap out content with Ajax. Then they want to share the URL. With the History API, we could have been changing the URL of the page right along with the user as they navigate, so the URL they see (and thus share or save) is relevant and correct.

#The Basics

To check out the features of this API it's as simple as heading into the Developer Tools and typing `history` into the console. If the API is supported in your browser of choice then we'll find a host of methods attached to this object:



These are the methods available to us to manipulate the browser's history.

We're interested in the `pushState` and `replaceState` methods in this tutorial. Returning to the console, we can experiment a little with the methods and see what happens to the URL when we use them. We'll cover the other parameters in this function later, but for now all we need to use is the final parameter:

```
history.replaceState(null, null, 'hello');
```

The `replaceState` method above switches out the URL in the address bar with '/hello' despite no assets being requested and the window remaining on the same page. Yet there is a problem here. Upon hitting the back button we'll find that we don't return to the URL of this article but instead we'll go *back* to whatever page we were on before. This is because `replaceState` does not manipulate the browser's history, it simply replaces the current URL in the address bar.

To fix this we'll need to use the `pushState` method instead:

```
history.pushState(null, null, 'hello');
```

Now if we click on the back button we should find it working as we'd like it to, since `pushState` has changed our history to include whatever URL we just passed into it. This is interesting, but what happens if we try something a little devious and pretend that the current URL wasn't css-tricks.com at all, but another website entirely?

```
history.pushState(null, null, 'https://twitter.com/hello');
```

This will throw an exception because the URL has to be of the **same origin** as the current one, otherwise we might risk major security flaws and give developers the ability to fool people into believing they were on a

different website altogether.

Returning to those other parameters that are passed into this method, we can summarise them like this:

```
history.pushState([data], [title], [url]);
```

1. The first parameter is the data we'll need if the state of the web page changes, for instance whenever someone presses the back or forwards button in their browser. Note that in Firefox this data is limited to 640k characters.
2. `title` is the second parameter which can be a string, but at the time of writing, every browser simply ignores it.
3. This final parameter is the URL we want to appear in the address bar.

#A Quick History

The most significant thing with these history API's is that they don't reload the page. In the past, the only way to change the URL was to change the `window.location` which always reloaded the page. Except, if all you changed was the `hash` (like how clicking a `<a href="#target">link</a>` doesn't reload the page).

This lead to the old hashbang method of changing the URL without a full page refresh. Famously, Twitter used to do things this way and was largely criticized for it (a hash not being a "real" resource location).

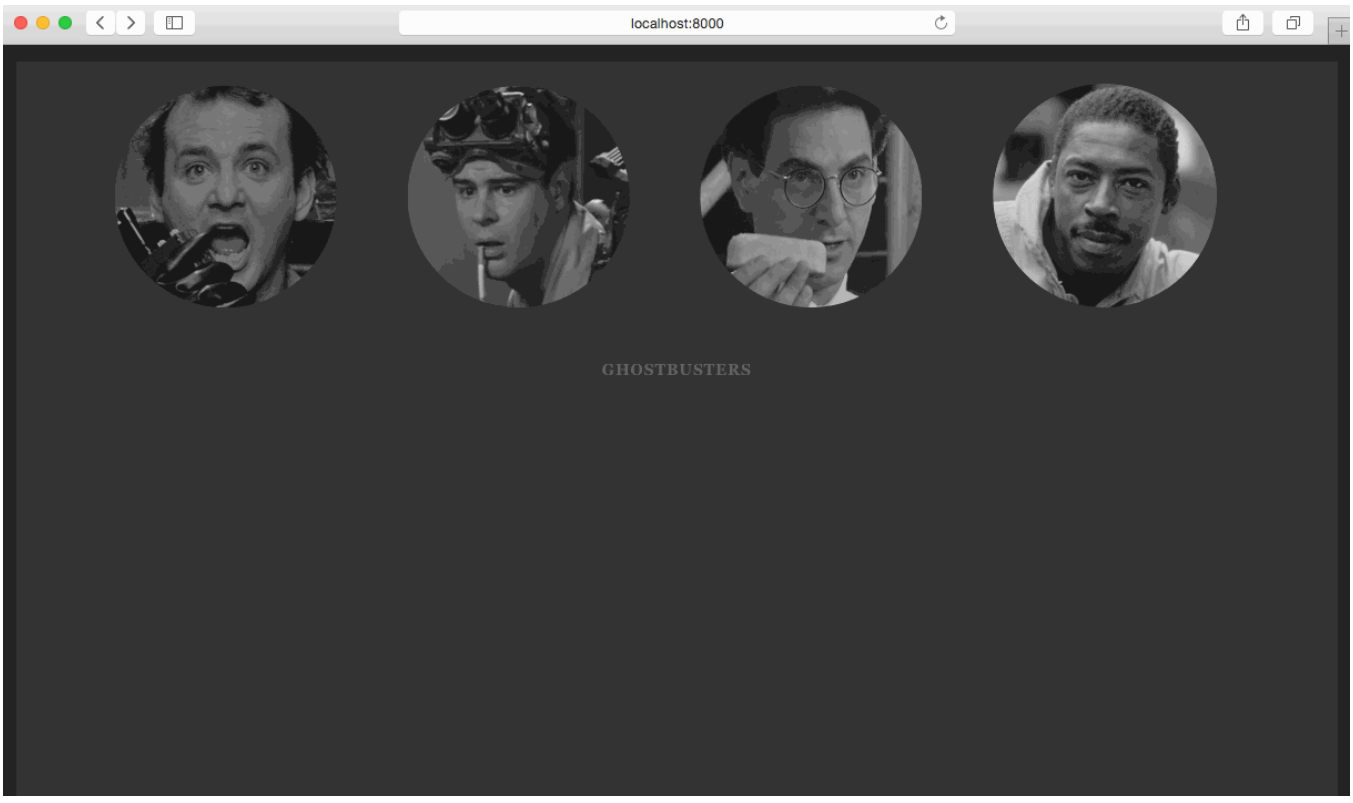
Twitter moved aways from that, and was one of the early proponents of this API. In 2012 the team described their new approach. Here they outline some of their problems when working at this kind of scale whilst also detailing how various browsers implement this specification.

#An example using pushState and Ajax

Let's build a demo!

In our imaginary interface we want the users of our website to find information about a character from Ghostbusters. When they select an image we need the text about that character to appear underneath and we also want to add a current class to each image so that it's clear who's been selected. Then when we click the back button the current class will jump to the previously selected character (and vice-versa for the forwards button) and of course we'll need the content beneath to switch back again, too.

Here's a working example that we can dissect:



The markup for this example is simple enough: we have a `.gallery` which contains some links and within each of them is an image. We then have the text beneath that we want to update with the selected name and the empty `.content` div that we want to replace with the data from each character's respective HTML files:

```
<div class="gallery">
  <a href="https://cdn.css-tricks.com/peter.html">
```

```
    <img src="bill.png" alt="Peter" class="peter" data-name="peter">
  </a>
  <a href="https://cdn.css-tricks.com/ray.html">
    <img src="ray.png" alt="Ray" class="ray" data-name="ray">
  </a>
  <a href="https://cdn.css-tricks.com/egon.html">
    <img src="egon.png" alt="Egon" class="egon" data-name="egon">
  </a>
  <a href="https://cdn.css-tricks.com/winston.html">
    <img src="winston.png" alt="Winston" class="winston" data-name="winston
  </a>
</div>

<p class="selected">Ghostbusters</p>
<p class="highlight"></p>

<div class="content"></div>
```

Without any JavaScript this page will still function as it should, clicking a link heads to the right page and clicking the back button also works just as a user would expect it too. Yay for accessibility and graceful degradation!

Next we'll hop on over to JavaScript where we can begin adding an event handler to each link inside the `.gallery` element by using event propagation, like so:

```
var container = document.querySelector('.gallery');

container.addEventListener('click', function(e) {
  if (e.target != e.currentTarget) {
    e.preventDefault();

  }
  e.stopPropagation();
}, false);
```

Inside this `if` statement we can then assign the `data-name` attribute of the image we select to the `data` variable. Then we'll append ".html" to it and

use that as the third parameter, the URL we'd like to load, in our `pushState` method (although in a real example we'd probably want to change the URL only *after* the Ajax request has been successful):

```
var data = e.target.getAttribute('data-name'),
  url = data + ".html";
  history.pushState(null, null, url);
```

(Alternatively, we could also grab the link's href attribute for this.)

I've replaced working code with comments so we can focus on the `pushState` method for now.

So at this point, clicking on an image will update the URL bar and the content with the Ajax request but clicking the back button won't send us to the previous character we selected. What we need to do here is to make another Ajax request when the user clicks the back/forwards button and then we'll need to update the URL once again with `pushState`.

We'll first head back and update the state parameter of our `pushState` method in order to stash that information away:

```
history.pushState(data, null, url);
```

This is the first parameter, `data` in the method above. Now anything that's set to that variable will be accessible to us in a `popstate` event which fires whenever the user clicks on the forward or back buttons.

```
window.addEventListener('popstate', function(e) {
```

```
});
```

Consequently we can then use this information however we like, which in this case is passing the name of the previous Ghostbuster we selected as a parameter into the Ajax `requestContent` function, which uses jQuery's `load` method:

```
function requestContent(file) {
  $('.content').load(file + ' .content');
}

window.addEventListener('popstate', function(e) {
  var character = e.state;

  if (character == null) {
    removeCurrentClass();
    textWrapper.innerHTML = " ";
    content.innerHTML = " ";
    document.title = defaultTitle;
  } else {
      updateText(character);
      requestContent(character + ".html");
      addCurrentClass(character);
      document.title = "Ghostbuster | " + character;
  }
});
```

If a user was to click on the picture of Ray our event listener would fire, which would then store the data attribute of our image within the `pushState` event. Consequently this loads the `ray.html` file which will be called upon if the user selects another image and then clicks the back button. *Phew*.

What does this leave us with? Well, if we click on a character and then share the URL we've updated, then that HTML file would be loaded instead. It might be a less confusing experience and we'll preserve the

integrity of our URLs whilst giving our users a faster browsing experience over all.

It's important to acknowledge that the example above is simplistic since loading content in this way with jQuery is very messy and we'd probably want to pass a more complex object into our `pushState` method but it shows us how we can immediately start learning how to use the History API. First we walk, then we run.

#The Next Step

If we were to use this technique on a larger scale then we should probably consider using a tool designed specifically for that purpose. For example pjax is a jQuery plugin that speeds up the process of using Ajax and pushState simultaneously, although it only supports browsers that use the History API.

History JS on the other hand supports older browsers with the old hash-fallback in the URLs.

#Cool URLs

I like thinking about URLs, and I particularly reference this post on URL design by Kyle Neath all the time:

> URLs are universal. They work in Firefox, Chrome, Safari, Internet Explorer, cURL, wget, your iPhone, Android and even written down on sticky notes. They are the one universal syntax of the web. Don't take that for granted. Any regular semi-technical user of your site should be able to navigate 90% of your app based off memory of the URL structure. In order to achieve this, your URLs will need to be pragmatic.

This means that regardless of any hacks or performance boosting tricks we might want to implement, web developers ought to cherish the URL and with the help of the HTML5 History API we can fix problems like the above

example with just a little elbow grease.

#Common Gotchas

- It's often a good idea to embed the location of an Ajax request in the `href` attributes of an anchor element.
- Make sure to `return true` from Javascript click handlers when people middle or command click so that we don't override them accidentally.

#Further Reading

- Mozilla's documentation on manipulating the browser history
- The Ajax gallery example from Dive into HTML5
- Twitter's implementation of pushState

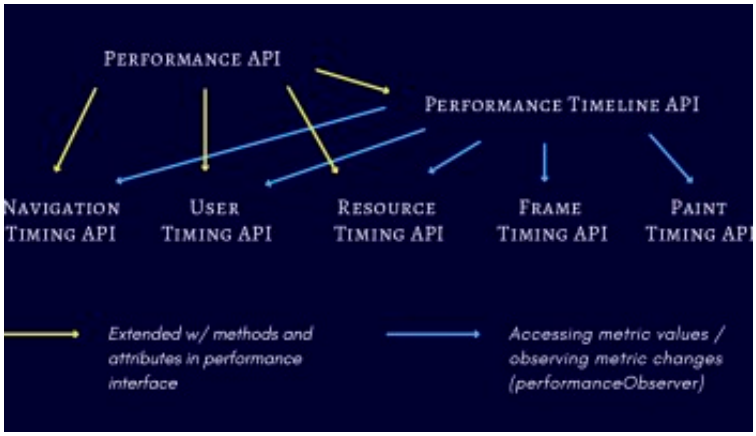
#Browser support

Chrome	Safari	Firefox	Opera	IE	Android	iOS
31+	7.1+	34+	11.50+	10+	4.3+	7.1+

#*Related*

Rethinking Dynamic Page Replacing Content

Jesse Shawl takes an old(ish) CSS-Tricks demo and updates it for today's world. Using the HTML5 history API he changes the URL and content of a page when navigation items are clicked, without refreshing the page.



Breaking Down the Performance API

JavaScript's Performance API is prudent, because it hands over tools to accurately measure the performance of Web pages, which, in spite of being performed since long before, never really became easy or precise enough. That said, it isn't as easy to get started with the API as it is to...

Everything You Need to Know About Instagram API Integration

The following is a guest post by Emerson This. This is a guide for web developers interested in integrating Instagram content on websites. It was only a few months ago when Instagram changed what was possible with their API, so this serves to explain that, what is possible now, and...