Extracted from:

Web Development Recipes
Second Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

2nd Edition

Web Development Recipes

Brian P. Hogan,
Chris Warren,
Mike Weber, and
Chris Johnson

edited by Rebecca Gulick

Web Development Recipes
Second Edition

Brian P. Hogan
Chris Warren
Mike Weber
Chris Johnson

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (index)
Eileen Cohen; Cathleen Small (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-056-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2015

Recipe 13

Extending Endless Pagination with pushState()

Problem

One of the things that makes the Internet great is that, from news articles to cat GIFs, everyone can easily share links with one another. But with more applications using Ajax, this is no longer the case by default; clicking an Ajax link no longer guarantees that the browser's URL is updated to reflect what the user is seeing. For many Ajax requests this is fine, but when large parts of the site change after a request, not keeping the URL up to date can cause issues. Not only does this prevent the sharing of links, but it breaks the back and refresh buttons.

Unfortunately, the endless-pagination code we wrote in Recipe 12, *Displaying Information with Endless Pagination* on page ?, has the same issue. As we scroll through the page and request new pages via Ajax, the browser's URL never updates to reflect the new content on the screen.

So, for example, if a user is on page 5 of a catalog and wants to share it with some friends via email, he'd likely copy and paste the browser's URL and say, "Check out this great deal!" Unfortunately, when the friends open the link they'll see page 1 and have no idea what he was talking about.

A user's own experience is also affected. When she clicks the back button on an all-Ajax site, she often ends up at whatever page led her to the site instead of the last Ajax page loaded. Then, frustrated, she clicks the forward button and ends up somewhere completely different. Thankfully, we have a solution for these common problems.

Ingredients

- jQuery
- Handlebars.js[13]
- QEDServer (for our test server)[14]

13. http://handlebarsjs.com/
14. A version for this book is available at http://webdevelopmentrecipes.com/.

Solution

We'll start with the code that we wrote in Recipe 12, *Displaying Information with Endless Pagination* on page ?, and finish implementing it. The old code works, but users can't easily share links with anyone. To keep our web karma in alignment and prevent user frustration, the right thing to do is to make this list page stateless. When we change the page that the user is looking at, we'll change the current URL to reflect these changes.

The HTML5 specification introduced a JavaScript function called pushState(), which lets us alter the URL without leaving the page. This is great news for web developers! We can make an entire Ajax web application that never goes through the traditional request/reload life cycle while behaving like a multi-page site. This means there's no need to re-request resources like images, style sheets, or JavaScript files every time we move to a new screen. And users can quickly share the current URL with others or use the refresh and back buttons as usual.

Using the pushState() Function

Although pushState() is widely implemented, some older browser versions don't support it. The available fallback solution relies on modifying the hash portion of a URL, but it's ugly—and it's not only an issue of having URLs that are displeasing to the eye. The Internet has a good long-term memory. Web pages may exist that include links that were added years ago, but the content has moved to a new server. If we use the URL hash as a stopgap for important information, we could be stuck supporting those deprecated links until the end of time. Since URL hashes are never sent to the server, our application would have to continue to read the URLs with JavaScript and redirect to the requested page.

With that said, let's see what it takes to make our endless products page stateless.

Parameters to Track

Because we don't know which page a user will load on the first request, we'll keep track of the starting page as well as the current page. If users go directly to page three, we want them to be able to get back to page three on subsequent visits. If they start scrolling down from page three and load multiple pages, for instance to page seven, we want to know that, too. We need a way to keep track of the start and end pages so that a hard refresh won't require the user to scroll through the site again.

Next we need a way to send the start and end pages from the client. The most direct way is to set these parameters in the URL during a GET request. When a page is first loaded, we'll set the page parameter of the URL to be the current page and assume the user wants to see only that page. If the client also passes in a start_page parameter, we'll know that the user wants to see a range of pages, from start_page through page. So following our earlier example, if the user is on page seven but started browsing from page three, our URL would look like this: http://localhost:8080/products?start_page=3&page=7.

This set of parameters should be enough information for us to re-create a list of products from the server and subsequently show users the same page they saw when they last visited this page:

statefulpagination/stateful_pagination.js

```
function getParameterByName(name) {
  var match = RegExp('[?&]' + name + '=([^&]*)')
    .exec(window.location.search);

  return match && decodeURIComponent(match[1].replace(/\+/g, ' '));
}

var currentPage = 0;
var startPage = 0;

function readParameters() {
  startPage = parseInt(getParameterByName('start_page'));
  if (isNaN(startPage)) {
    startPage = parseInt(getParameterByName('page'));
  }
  if (isNaN(startPage)) {
    startPage = 1;
  }
  currentPage = startPage - 1;

  if (getParameterByName('page')) {
    endPage = parseInt(getParameterByName('page'));
    for (i = currentPage; i < endPage; i++) {
      getNextPage(true);
    }
  }

  observeScroll();
}
```

All we're doing here is figuring out the start_page and current_page and then requesting those pages from the server. We use mostly the same function from the previous chapter, getNextPage(), but it's been slightly modified to allow multiple requests at a time:

```
var loadingPage = 0;
function getNextPage(ignoreMutexBlocking) {
  if (!ignoreMutexBlocking && loadingPage != 0) return;

  loadingPage++;
  $.getJSON(nextPageWithJSON(), {}, updateContent).
    complete(function() { loadingPage-- });
}
```

Normally when the user is scrolling we want to prevent multiple, overlapping requests. But right now it's all right, since we know exactly which pages should be requested, so we'll pass in true to ignore the mutex block. Then we want to call the readParameters() function when the page loads to set the initial state of the page:

```
readParameters();
```

Just as we tracked the currentPage in the code on page ?, we want to track the startPage. We'll grab this parameter from the URL so we can make the requests for the pages that haven't been loaded yet. This number will never change, but we do want to make sure that it gets added to the URL and stays there every time a new page is requested.

Updating the Browser's URL

To update the URL, let's write a function called updateBrowserUrl() that'll call pushState() and set the parameters for the start_page and page. It's important to remember that not every browser supports pushState(), so we need to check that it's defined before we can call it:

```
function updateBrowserUrl() {
  if (window.history.pushState == undefined) return;

  var newURL = '?start_page=' + startPage + '&page=' + currentPage;
  window.history.pushState({}, '', newURL);
}
```

The pushState() function takes three parameters. The first allows us to track any state we want with a JSON object. This argument could potentially be a storage point for information we want the browser to remember that doesn't make sense to have in the parameters, such as the JSON we've already received from the server when we scrolled. But since our data is relatively lightweight and easy to get from the server, we skip this. For now we'll pass in an empty hash. The second argument is the title of the page. This feature isn't widely implemented yet, and for our purposes, even if it were, we don't have a reason to update this page's title. We pass in a filler argument again; this time an empty string.

Now we get to the meat, or the tofu if you're vegetarian, of the pushState() function. The third parameter is how we want the URL to change. This method is flexible and can be either an absolute path or only the parameters to be updated at the end of the URL. For security reasons, we can't change the domain of the URL, but we can change everything after the top-level domain with relative ease. Since we're worried only about updating the parameters of the URL, we prepend the pushState()'s third parameter with a question mark (?). Finally, we set the start_page and page parameters, and if they already exist, pushState() is smart enough to update these parameters for us.

Lastly, we add a call to updateBrowserUrl() from the updateContent() function to make our endless pagination code state-aware:

statefulpagination/stateful_pagination.js

```
function updateContent(response) {
  loadData(response);
  updateBrowserUrl();
}
```

Now our users can use the back button to leave our page and return with the forward button without losing their spot. They can also hit the refresh button with impunity and get the same results. Most important, our links are now sharable across the web. We've been able to make the URL for our index page behave like a traditional non-Ajax site with minimal effort, thanks to the hard work of modern browser developers.

Further Exploration

As we add more JavaScript and Ajax to our pages, we have to be aware of how the interfaces behave. HTML5's pushState() method and the History API give us the tools we need to provide support for the regular controls in the browser that people already know how to use. Abstraction layers like History.js[15] make it even easier to use and provide graceful fallbacks for browsers that don't support the History API.

The approaches we discussed in this recipe are regularly being used by frameworks like Backbone.js, which means even easier back button support for the most complex single-page applications.

Also See

- Recipe 11, *Rendering HTML with Handlebars Templates* on page ?
- Recipe 13, *Extending Endless Pagination with pushState()* on page 5

15. https://github.com/browserstate/history.js