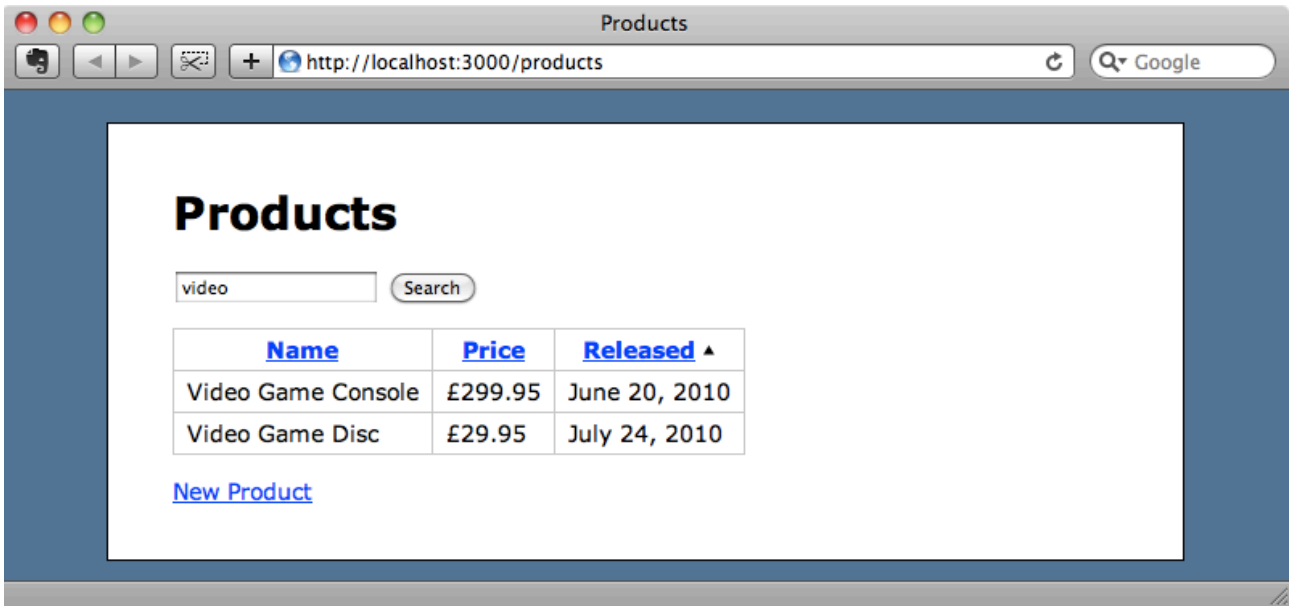
Episode 246

AJAX History State

In episode 240 [watch[1], read[2]] we showed you how to use AJAX to add sorting, pagination and searching to a table of products. One of the problems with using AJAX like this is that the page's URL doesn't change when we sort or filter the table. This means that we can't use the back button to go back through the changes we've made to the table, nor can we bookmark a sorted or filtered page so that we can come back to the same results we had before.



This is a common problem when AJAX functionality is added to a site and it would be useful if we could find a solution. We'll show you how to do this in this episode.

Introducing the history.pushState method

We covered this topic in episode 175 [watch[3], read[4]] where we added history and bookmarking to an AJAX-enabled app by changing the URL. In that episode when we paged through the results an anchor was added to the URL so that for, say, page 3, the URL was `http://localhost:3000/#page=3`. This enables us to bookmark pages and makes the back button work as expected. This approach works well but quickly becomes complicated in more complex scenarios such as we have here where we're sorting and searching as well as paging through results.

---

[1] http://railscasts.com/episodes/240-search-sort-paginate-with-ajax

[2] http://asciicasts.com/episodes/240-search-sort-paginate-with-ajax

[3] http://railscasts.com/episodes/175-ajax-history-and-bookmarks

[4] http://asciicasts.com/episodes/175-ajax-history-and-bookmarks

The technique we're going to use here is used by GitHub in its file browser. When we click on a folder in a GitHub repository in a browser the page updates through AJAX and the whole URL is updated by JavaScript, not just the anchor tag. This is a much nicer solution and it means that we can reload or bookmark the page. Also the page is added to the browser's history so the back button will work as we expect it to.

GitHub blogged about this[5] and the post is well worth reading to see how this was implemented. To see it in action you'll need a recent version of Safari, Chrome or Firefox, but it's a technique definitely worth making use of when you can. The "magic" that makes this work are the `history` object's `pushState` and `replaceState` functions and the `popstate` event. There are several useful links on the GitHub blog post that give more details about these functions and demos showing them in use.

Let's see how easy it is to add this functionality to our simple shopping application so that it works when we sort, page or search the table of products. The JavaScript we added to the original application looks like this:

/public/javascripts/application.js

```
$(function () {
  $('#products th a, #products .pagination a').live('click',
    function () {
      $.getScript(this.href);
      return false;
    }
  );

  $('#products_search input').keyup(function () {
    $.get($('#products_search').attr('action'), ↵
      $('#products_search').serialize(), null, 'script');
    return false;
  });
})
```

[5] https://github.com/blog/760-the-tree-slider

If the code above contains things that you don't find familiar then it's worth taking a look at episode 240 for more details. We're using jQuery here but this technique will work with Prototype too as far as using `pushState` and `replaceState` go.

The first part of the code above handles the AJAX functionality for sorting and pagination. We'll add a call to `history.pushState` in this code so that the URL is added to the history when one of the sorting or pagination links is clicked.

There is some useful documentation about `pushState` on Mozilla's developer site[6] which shows that it takes three parameters. The first is a state object, which can be anything and which will be returned when the `popState` event is fired. The second argument is a title, while the third is a URL. Now that we know this we can add the `pushState` call.

/public/javascripts/application.js

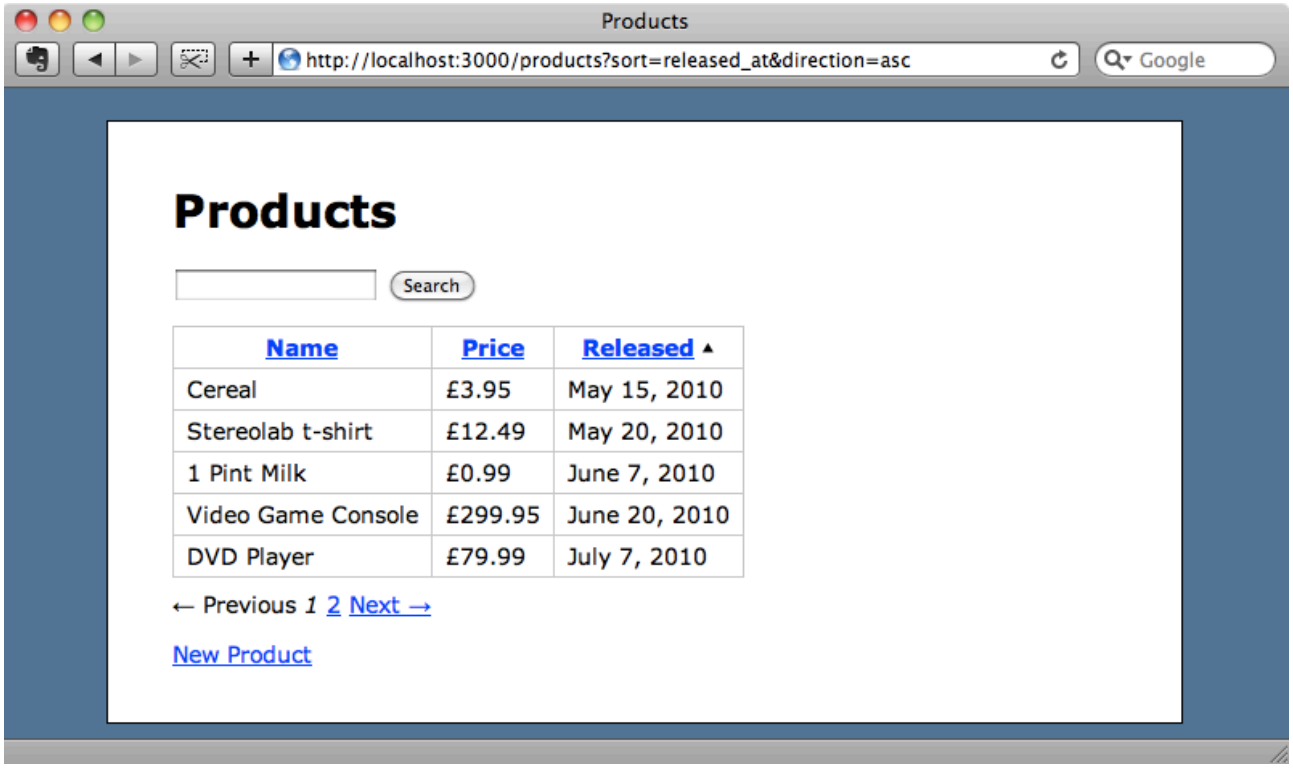
```
$('#products th a, #products .pagination a').live('click',
  function () {
    $.getScript(this.href);
    history.pushState(null, "", this.href);
    return false;
  });
```

We don't need to store any state here so we pass `null` as the first parameter. Likewise we don't need a title so we pass in an empty string. For the URL we pass in `this.href` which is the URL of the link that is clicked.

If we reload the page now and click one of the AJAX sorting or pagination links the table will update without the page reloading, but now each time we click a link the URL will change in the address bar to whatever we passed into the `pushState` function and the URL is added to the browser's history.

---

[6] https://developer.mozilla.org/en/DOM/Manipulating_the_browser_history#Adding_and_modifying_history_entries



So, now we can sort and paginate the table and bookmark it and when we come back to it or reload it the pagination and sorting are preserved, even though the page is being update dynamically through AJAX. The back button, however, does not work as we'd expect yet as we're not listening for the `popstate` event. We can fix this by firing a function when the `popstate` event fires.

/public/javascripts/application.js

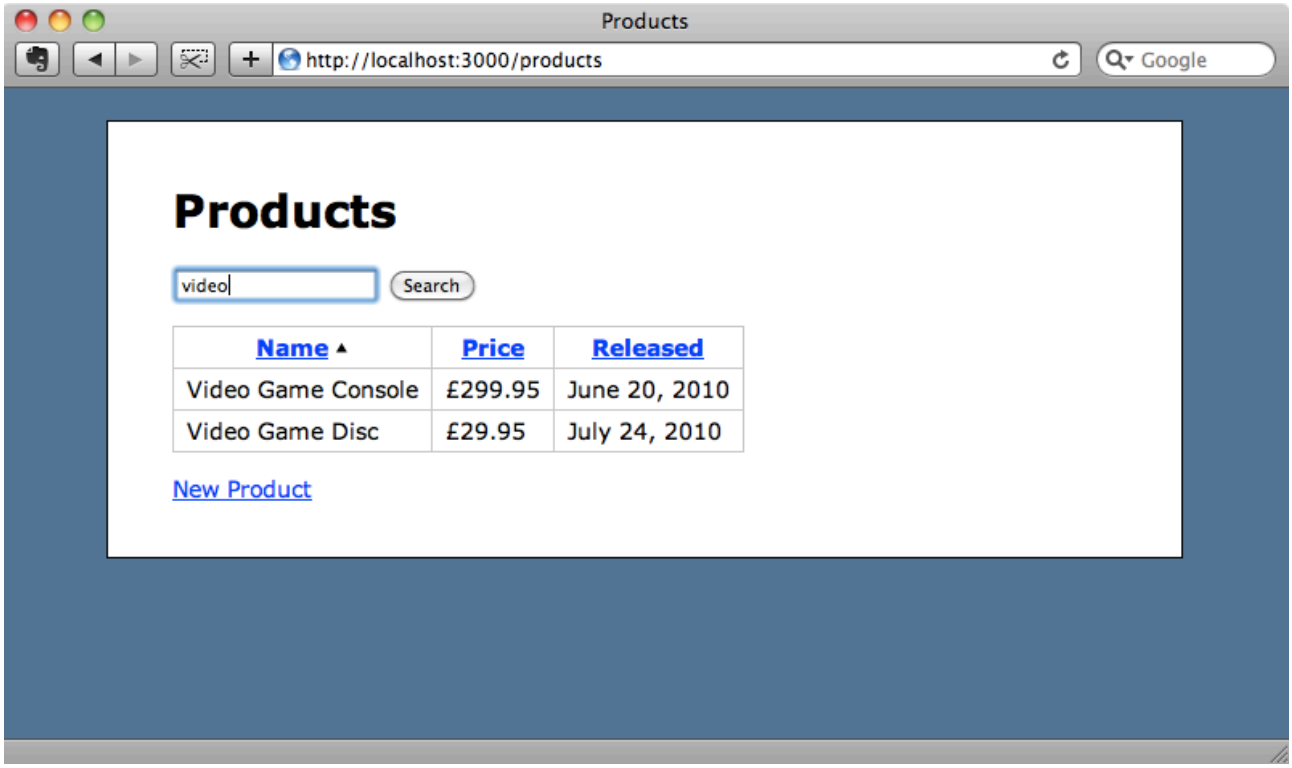
```
$(function () {
  // Other functions omitted.
  $(window).bind("popstate", function () {
    $.getScript(location.href);
  });
})
```

By the time the `popstate` event fires the URL will have already changed back to the previous one in the history and so we can update the products table by calling `$.getScript` and passing in the current URL.

If we reload the page now and sort or paginate the table a few times we'll find that the back button now works and that we can navigate back through the changes we have made.

Searching

We'll take a look at the search functionality next. If we search for a product with our application now, it will filter the results, but the URL won't change so if we bookmark or reload the page after making a search we lose the search.



The JavaScript for the AJAX searching is this:

/public/javascripts/application.js

```
$(function () {
  // Other functions omitted.
  $('#products_search input').keyup(function () {
    $.get($('#products_search').attr('action'), ↵
      $('#products_search').serialize(), null, 'script');
    return false;
  });
})
```

We'll use `pushState` here in the same way we did when we modified the sorting code.

/public/javascripts/application.js

```
$(function () {
  // Other functions omitted.
  $('#products_search input').keyup(function () {
    var action = $('#products_search').attr('action');
    var formData = $('#products_search').serialize();
    $.get(action, formData, null, 'script');
    history.pushState(null, "", action + "?" + formData);
    return false;
  });
})
```

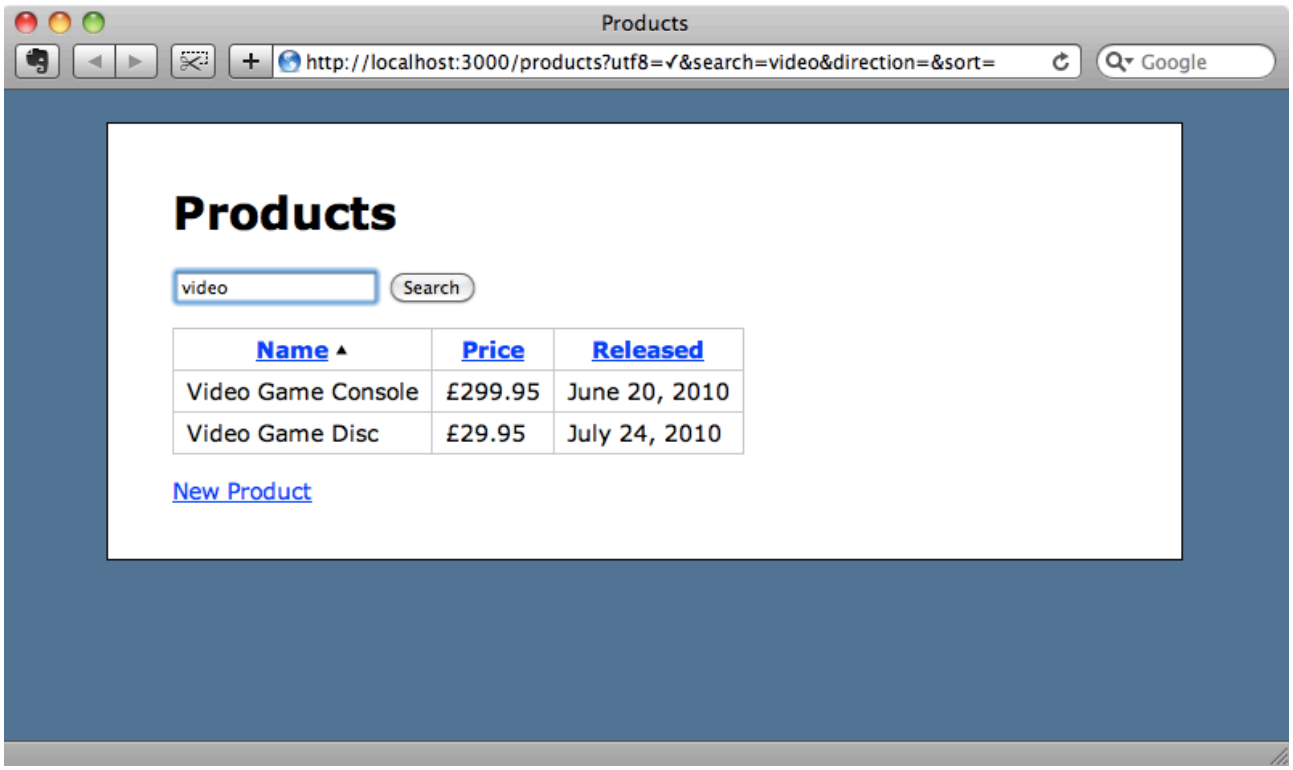
The URL that `pushState` will record here is a combination of the search form's action and the serialized form data, joined together with a question mark to make a valid URL.

When we reload the page now we'll see that the URL now changes every time we type a key in the search field. When we hit the back button, however, there's a page in the history for each key we've entered in the search term, which is not ideal. This is because we're calling `pushState` every time a key is pressed. It would be better if we could replace the current state each time a key is pressed rather than adding a new state. Fortunately we can do this very simply by changing `pushState` in the code above with `replaceState`. This will replace the current state instead of adding another one each time the function is called.

/public/javascripts/application.js

```
$(function () {
  // Other functions omitted.
  $('#products_search input').keyup(function () {
    var action = $('#products_search').attr('action');
    var formData = $('#products_search').serialize();
    $.get(action, formData, null, 'script');
    history.replaceState(null, "", action + "?" + formData);
    return false;
  });
})
```

If we open the products page in a new window now and search for something we'll see that the URL changes but that the changes aren't added to the browser's history.



This isn't quite the functionality we want as ideally we'd use `pushState` when the user starts searching and then `replaceState` subsequently so that the search term is added to the history. This is a little outside the scope of this episode, however, so we won't be covering that here.

Adding a Title To Each State

If we sort the products a few different ways and then click and hold the back button to see the browser's history it will show each page's URL which can make it difficult to choose the page we want to go back to. It would make the history more usable if a title was displayed for each page instead. We can do this by setting the second argument in each `pushState` call to be the page's title.

How do we set the page title to be something useful? What we'll do is set the `title` for each page and then use that title in `pushState`. We can do that by setting a title in the `index.js.erb` file that sends back the updated table when an AJAX call is made.

/app/views/products/index.js.erb

```
$('#products').html('<%= escape_javascript( ↵
  render("products")) %>');
document.title = "<%= escape_javascript("#{ ↵
  params[:search].to_s.titleize} Products by #{(params[:sort] ↵
  || 'name').titleize} - Page #{(params[:page] || 1)}") %>"
```

We'll set the title so that it shows the search and sort terms and the current page so that all the relevant information is available. We can then update the calls to `pushState` and `replaceState` so that they set the title to the page's title.

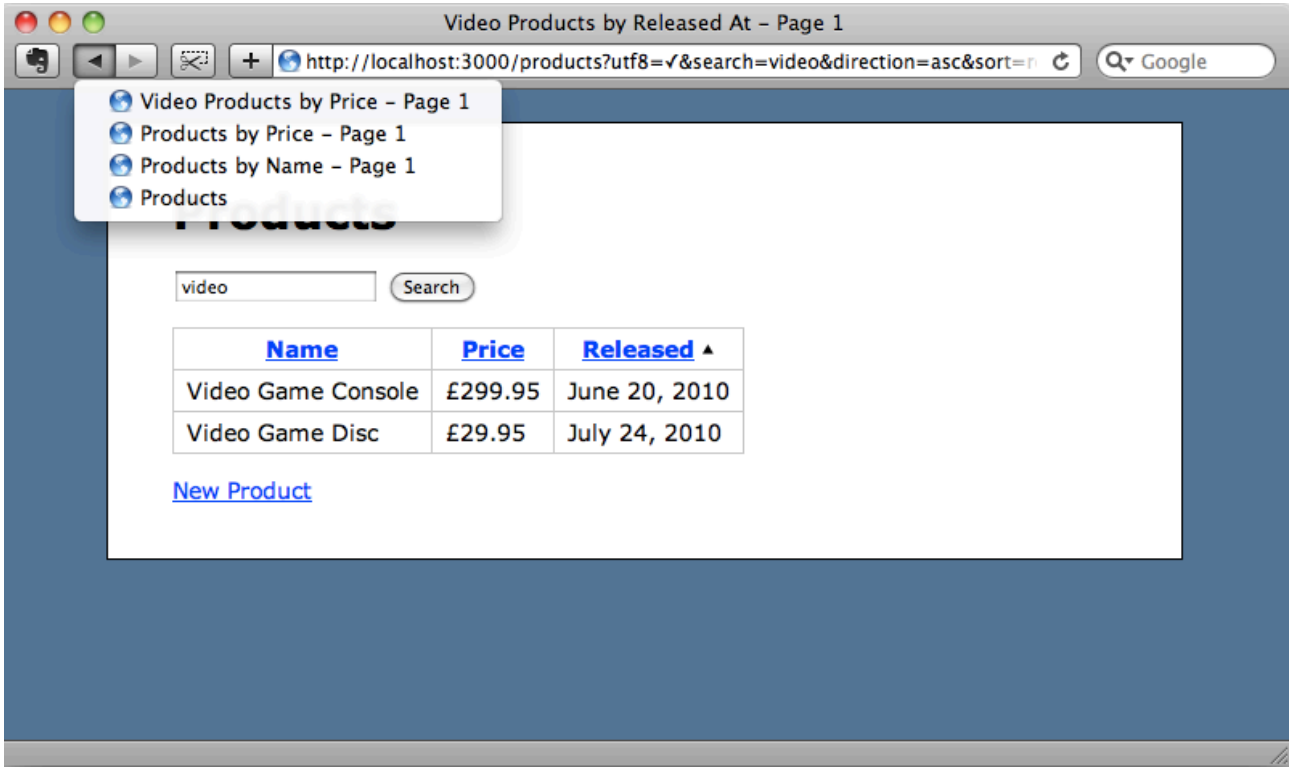
/public/javascripts/application.js

```
$(function () {
  $('#products th a, #products .pagination a').live('click',
function () {
    $.getScript(this.href);
    history.pushState(null, document.title, this.href);
    return false;
  });

  $('#products_search input').keyup(function () {
    var action = $('#products_search').attr('action');
    var formData = $('#products_search').serialize();
    $.get(action, formData, null, 'script');
    history.replaceState(null, document.title, action + "?" + ↵
      formData);
    return false;
  });

  $(window).bind("popstate", function () {
    $.getScript(location.href);
  });
})
```

Now, each time we search or sort the table of products the page's title will change and if we look at the browser history we'll see a list of titles rather than URLs.



Handling Older Browsers

Our application's new functionality works pretty well but we've been assuming that the `history.pushState` method is available in the user's browser which, given that only the newest versions of Safari, Chrome and Firefox support it is quite likely. To handle all browsers we should first check that the browser supports `pushState` and modify our application's behaviour to suit.

What we'll do is check that the `history` object and `history.pushState` method exist and if not we'll disable all of the AJAX functionality so that the application falls back gracefully to using HTML links.

/public/javascripts/application.js

```
if (history && history.pushState) {
  $(function () {
    $('#products th a, #products .pagination a').live('click',
function () {
      $.getScript(this.href);
      history.pushState(null, document.title, this.href);
      return false;
    });

    $('#products_search input').keyup(function () {
      var action = $('#products_search').attr('action');
      var formData = $('#products_search').serialize();
      $.get(action, formData, null, 'script');
      history.replaceState(null, document.title, action + "?" +
formData);
      return false;
    });

    $(window).bind("popstate", function () {
      $.getScript(location.href);
    });
  })
}
```

That's it for this episode. While it hasn't been Rails-specific it has dealt with issues that you'll encounter when dealing with AJAX in Rails applications. Being able to change the URL like this enables us to treat AJAX links at the same level as traditional HTML ones.