Greasemonkey:

The web is a wonderful thing, but sometimes it doesn't work exactly as you want it to. **Juliet Kemp** and Greasemonkey can help you fix that.

The idea behind *Greasemonkey* is pretty simple. It's a *Firefox* extension, installed in the same way as any other *Firefox* extension (find it via the Tools > Addons menu and hit Install). However, it doesn't do anything in and of itself: what it does is to enable you to run scripts, either by other people or by yourself, which will alter the way webpages look and function.

Greasemonkey user scripts are the bits of code that actually do the work – *Greasemonkey* itself just loads and manages these. User scripts are written in JavaScript, but be warned: for security reasons, this isn't just a question of writing regular JavaScript and away you go. There are some gotchas to be aware of, although the scripts in this tutorial don't encounter any of them.

A quick note if you're unfamiliar with JavaScript: this tutorial isn't going to explain JavaScript syntax in any detail, but don't let that stop you from giving it a go. It's all fairly logical and the code snippets are all explained.

To install a script that someone else has written, you navigate to its location in *Firefox* and click on the link to the script. You'll get an install popup, as with a normal extension, and can either look at the source code of the script first or, if you're feeling trusting, just install it.

Our expert

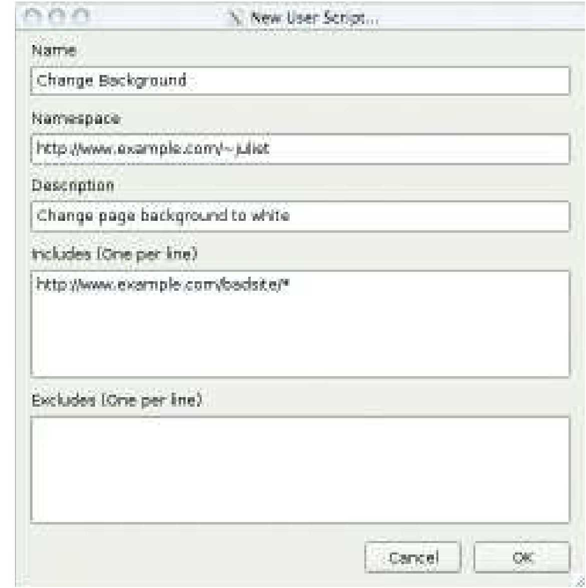
Juliet Kemp is a sysadmin and writer who spends a large portion of her time on the web and is quite often hit by the urge to stick a spork in her eyes. *Greasemonkey* helps avoid this happening.

Part 1 My First Greasemonkey Script

Greasemonkey provides a helpful dialog to make writing a script as straightforward as possible. In your *Firefox* window, once *Greasemonkey* is installed, there'll be a little monkey face in the right-hand corner of the status bar. Right-click on that and you'll get a menu that includes the option "New User Script". Click that and you'll get a dialog looking a bit like the box on the right.

The 'name' is just the name of your script – it's best to choose something that obviously indicates what it does, for ease of script management later on. The 'namespace' is to avoid your script clashing with others. If you try to install a script that has the same name as an already-installed one, it's the namespace that governs whether it will overwrite the old one (if the namespace is the same) or co-exist with it (if they're different). There are a couple of things you can do here: the first one is to use your own website as the domain name. Alternatively, you can use **http://localhost**, or if you're intending to upload it to **http://userscripts.org** when you're done, you can use that. Current versions of *Greasemonkey* won't allow you to leave it blank.

'Description' is for a human-readable line describing what the script does. It's a very good idea to fill this field in, even for your own scripts – you may wind up with stacks of the things and they'll be a lot easier to manage if you provide extra clues about which is which.



The New User Script dialog in all its blank glory. Enter as much information here as you can for an easier future.

LXF118.tut_grease 90 12/3/09 1:06:23 pm

Hack the web

The 'include' and 'exclude' rules govern on which sites a script will run, and can include wildcards. So, **www.example.com/*** will match **www.example.com/** and all pages starting with that URL (whereas **www.example.com/** without the asterisk will just include the front page). You can also use wildcards for parts of names: **http://*.example.com/f*** will match any page whose path begins with f, on any server in the **example.com** domain. By default, the include box will contain the page you were on when you clicked the new script option, but you're free to delete that. If an include rule is matched *and* no exclude rule is matched, the script will run. If you have no include rule, *Greasemonkey* assumes **@include ***, ie, that every URL is matched, so the script will run on every page you load.

This first script is going to set the background of a page to white – very useful indeed if you come across a page whose author has a fondness for eyeball-searing pink, or the sort of repeating background image that generates a headache within seconds. So pick a website you want to change the background of and put it in the **@include** box (here I'm using **www.example.com**), and set the other fields as appropriate.

Once you've filled this in, you will be asked for your preferred editor (if there's not already one set), and then *Greasemonkey* will load the script file – which currently will just contain the metadata – up in your editor, ready for you to write something.

At this point, the code you're faced with will look a fair bit like this:

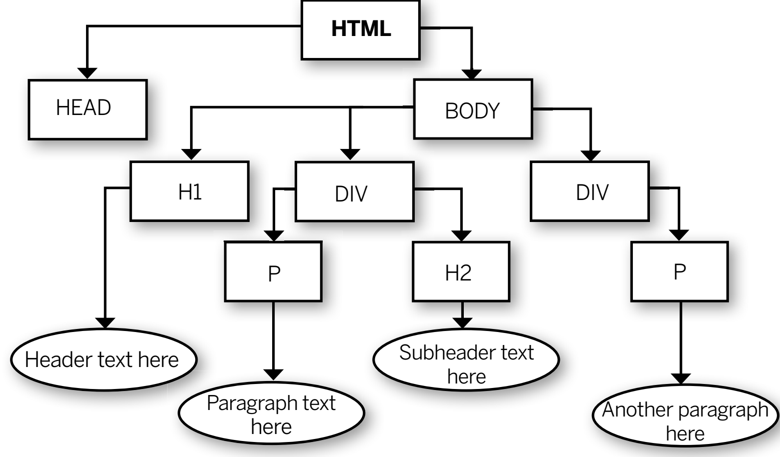
```
// ==UserScript==
// @name   Background Change
// @namespace  http://www.example.com/~juliet/
// @description Change the background colour of a page
// @include  http://www.example.com/*
// ==/UserScript==
```

Now it's time to actually write the script. All this first script does is to change the background colour of any pages from sites in the **include** domain to white. (There really are some unpleasant background colour choices out there.) For a page without frames or other complications, this is very straightforward: just a single line.

```
document.body.style.background = "#ffffff";
```

document is the built-in way of referring to the current page. It's a DOM (Document Object Model) object that represents the entire HTML document. Think of this as a tree of HTML elements seen as objects, with each new element branching off as a 'child' of the one before it – have a look at the diagram above-right, which shows a possible structure for the body part of an HTML document.

The notation for referring to an object in this model is **toplevel.child.childofchild**. So this line takes the document, then the body element, then the style of the body, then the background attribute of the style... and sets it to white.

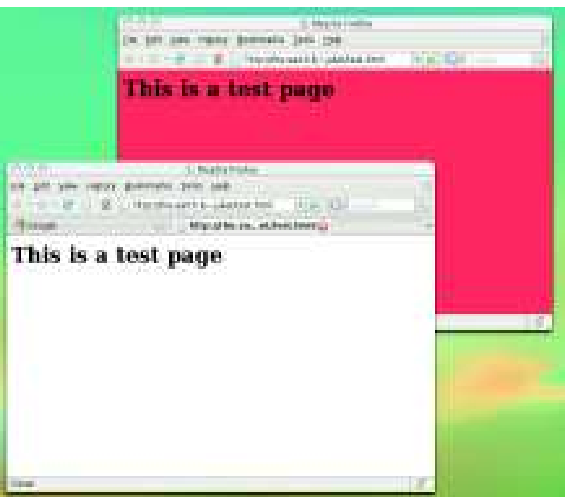


› **An HTML document as a DOM tree – each child node branches down from its parent node.**

(**#ffffff** is white in hexadecimal notation, which is one of the HTML standards. You could also just use **white**.)

Try it out now – pick a page with a non-white background, use the Manage Scripts menu to add that to the includes for your script and reload the page. When testing, remember: you're not actually doing anything to the web page you're editing. You're just changing it for you. So if you do something catastrophic, no problem! You can just turn your script off, or edit it and reload the page. So feel free to experiment.

When you're testing, if you left-click on the little monkey face, it'll toggle *Greasemonkey* on/off. So you can toggle it off, check how the page looks currently, toggle it on, reload and see what your script is doing. »



› **The page with a horrible background and with a nice white background. Note the little monkey face in the bottom right-hand corner!**

Quick tip

When writing your include rules, using the Magic TLD syntax **.tld** will match against any top-level domain (including a list of two-level domains intended to match **.co.uk** and so forth). So **example.tld** would match **example.com**, **example.co.uk**, **example.org**, and a whole set of other domains. However, for security reasons, this shouldn't be used if your script deals with private information.

» **If you missed last issue** Call 0870 837 4773 or +44 1858 438795.

Part 2 Set CSS styles

This script won't, however, work with frames; it'll only change the main body background style. If your eye-bleedingly backgrounded page has frames, your best bet to hit all of them is to insert a CSS style that overrides the existing one. To edit the existing file, right-click the monkey face and choose 'Manage User Scripts'. Select your script and click Edit to bring it up again in your chosen text editor. While you're testing, you don't need to close the editor or the Manage User Scripts dialog – just save the file and try reloading the page.

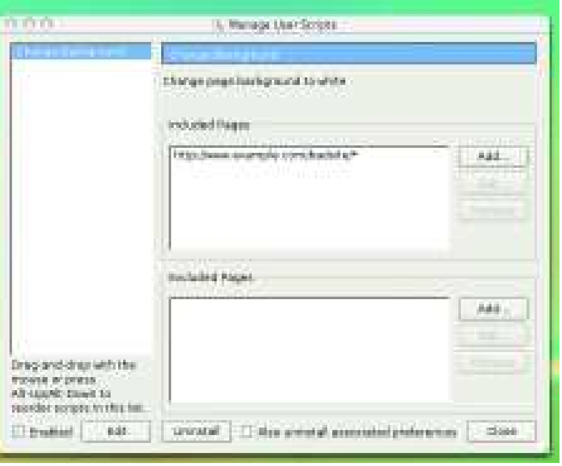
```
function addCss(cssString) {
        var head = document.
getElementsByTagName('head')[0];
        return unless head;
        var newCss = document.createElement('style');
        newCss.type = "text/css";
        newCss.innerHTML = cssString;
        head.appendChild(newCss);
}
addCss (
        '* { background-color: #ffffff ! important; }'
);
```

The first function (**addCss**) sets up a way of adding a global CSS file to the page. As before, **document** refers to the current page, but in this instance we're using the **getElementsByTagName** function (which does what it says on the tin) to get the first **head** element. For those of you who aren't entirely familiar with CSS: you set your CSS up in the **head** section of an HTML file, so we need to get that element to act as a parent to our new CSS element. There's an error-catching line in there (return without doing anything if there's no head section), then the script creates a **style** element and gives it the **text/css** type. The line

```
newCss.innerHTML = cssString
```

is where the function takes whatever is passed into it, and pastes that into the style element. Then the fully created style element gets added to the **head** element and we're done.

The final couple of lines are the bit that actually call the function, with the argument that sets the background colour



› **The Manage User Scripts dialog. You can edit the include/exclude rules here without needing to open the script up.**

– this goes where **cssString** is. Note: those are regular brackets (), because you're calling a function, not setting one up. The **! important** flag ensures that your CSS styles override those of the page itself.

Effectively, what this does is to add these lines to the header of your HTML page:

```
<style type="text/css">
        background-color: #ffffff ! important;
</style>
```

You can use this technique to set your own CSS preferences for anything else – just make that CSS string, when you call the **addCss** function, say what you want, eg:

```
addCss (
        '* { background-color: #ffffff ! important;
                text-align: center ! important;
                color: black ! important; }'
);
```

to make the background white, and all the text centred and in black.

Quick tip

As shown in the DOM diagram earlier, a 'child' of an HTML element is an element that's contained within another one. Imagine the document as a big tree, with every open-tag (eg **<p>** starting a new branch, and every close-tag (eg **</p>**) closing that branch. Self-closing tags (eg **
) make a branch, but can't have any children. So you could restrict your search within a particular paragraph or **div section.

Part 3 Alter specific elements

OK, so now you can change a single style aspect by changing the DOM object, or alter lots of style aspects by putting in your own CSS; and you can make that site-specific via the include/exclude rules that are built into *Greasemonkey*. What about when you want to find a particular type of element that occurs multiple times on a page, and alter it? For example, maybe you're using a forum where sometimes people have non-work-friendly user icons, so you want to just replace all the images with a nice safe image that you have somewhere on your own webspace. So, first of all, let's find all the images on the page:

```
var allImgs,thisImg;
allImgs = document.evaluate('//img[@src]',
  document,
  null,
  XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
  null);
```

The good bit here is the **document.evaluate** method. Getting the hang of this will give you an enormous amount of power. The first parameter (**'img[@src]'**) is an XPath query (of which more in a moment); the second is the element that you want to search. In this case that's the whole document, but you could restrict it further, and the query will only look through elements that are children of that element. The third parameter is a namespace resolver function, which is only relevant to **application/xhtml+xml** type pages.

The fourth parameter is how you want your results returned. The type here gives them back in random order, which is usually fine; if for some reason you want them back as they occur in the page, use **XPathResult.ORDERED_NODE_SNAPSHOT_TYPE** instead. The final parameter allows you to merge query results – pass in the result of a previous **document.evaluate** call and it'll merge the two together. Feel free to experiment with this!

» **Never miss another issue** Subscribe to the #1 source for Linux on p102.

The XPath query (the first parameter) is the powerhouse of this function. XPath is a powerful XML query language which is built in to *Firefox* and which you can therefore use from within *Greasemonkey*. If you want to find a particular set of elements, you could crawl through the DOM tree, retrieving sets of nodes and searching through them for whatever it is you're after. However, this is pretty slow and a bit ugly in terms of code. XPath gives you a quicker and cleaner way of finding pretty much anything you care to specify in a web page. I'll use it in a couple of scripts here, which will hopefully give you an idea of how it works – check out the specification online or one of the available tutorials if you want to know more about this. It's really very flexible: if you can think of a set of results you want to get out of an HTML document, you can probably construct an XPath query that will return them for you.

Back to our script: now you've got your images, you want to do something with them. This next piece of code goes in the same script, after the bit above:

```
for (var i=0;i<allImgs.snapshotLength;i++) {
 var thisImg = allImgs.snapshotItem(i);
 var src = thisImg.src;
            var srcMatch = src.match('^http://www.example.com/forums/userpic/');
            if (srcMatch != null) {
  thisImg.src = 'http://www.example.com/~juliet/safepic.gif';
            }
}
```

snapshotLength and **snapshotItem** are provided methods that work on the result of a **document.evaluate** call and give you, respectively, the total number of items returned, and a particular item. So you can stick both of those into a **for** loop, as here, and iterate over every result (here, every image on the page) returned by your XPath query.

A quick note: in regular JavaScript, you could iterate over this collection like so:

```
for (var thisImg in allImgs) {
           // do stuff
}
```

Because of the way that *Greasemonkey* implements script security, this won't work in a *Greasemonkey* script. You need to do it the long way around.

thisImg.src gives you the value of the **src** attribute of the image. So if your image tag were **<img src="foo.jpg">**, **thisImg.src** would return **foo.jpg**. (You can access the **width** or **height** attributes, or any other **img** attribute, with a similar syntax. Check out DOM object references online for more information.)

The final section tries to match the **src** value with the value expected for userpics on this forum (to find this for your own forum, you would need to take a look at the source code or a page of it), and if the result is non-null (indicating that there is a match), it replaces the image src value with your safe image. (You could write this in one fewer line by putting the **src.match** call in there directly, but this is a little easier to read.) And we're done!

Quick tip

The original version of *Greasemonkey* had some major security holes, arising from the way in which it injected user scripts directly into webpages, which could enable malicious pages to get access to your scripts. These days, *Greasemonkey* operates differently, effectively running everything in a sandbox and using wrappers to access objects on the remote web page that your script wants to alter.

Part 4 Replace text

Next up, let's take a look at replacing text that occurs anywhere in a webpage. Perhaps you are particularly fed up with the word "incentivize" occurring repeatedly in your corporate homepage, which, distressingly, you are required to look at on a regular basis. Let's replace it with "pling" instead (or any other word you find mildly amusing, rather than eyeball-spork-inducing).

```
// ==UserScript==
// @name   Deincentivize
// @namespace  http://www.example.com/~juliet/
// @description Replace "incentivize" on corporate homepage
// @include  http://www.example.net/corporatehome
// ==/UserScript==
textNodes = document.evaluate(
  "//text()",
  document,
  null,
  XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
  null);
var searchRE = new RegExp('incentivize','gi');
var replace = 'pling';
for (var i=0;i<textNodes.snapshotLength;i++) {
  var node = textNodes.snapshotItem(i);
  node.data = node.data.replace(searchRE, replace);
```

The first section of code should be familiar from our earlier script. It looks for any text nodes in the document. Next we set up the regular expression. The constructor **new RegExp()** takes two arguments. The first is the string that you're looking for. The second is the modifier. **g** means global match – replace every occurrence of the string, not just the first one. (Most of the time, you'll want to use this.) **i** makes the match case-insensitive. There's also **m** for multi-line mode, which makes the start-of-line and end-of-line anchors (**^** and **$**) match before and after newlines rather than matching the ends of the text in the node.

Finally, there's another for loop to do the real work of going through the XPath query output, looking for our search string, and replace it with our replace string. Easy!

There's a vast amount more you can do with *Greasemonkey* – the key is to start experimenting and see what changes you can make. Have fun remoulding the web to your own preferences! LXF

Debugging fun

In an ideal world, everything you ever write would work exactly as planned first time. Good luck with that.

For those of us who occasionally don't get it right the first time around, here are a couple of debugging tools to help you out:

1 DOM Inspector and InspectThis – available as add-ons for *Firefox 3*. The DOM Inspector (started from the Tools menu once you've installed it) enables you to take a look at the Document Object Model of a page – the structure of the page. InspectThis allows you to look at a particular page element by right-clicking on it and choosing "Inspect Element" from the context menu. Use both of these to get information about what node names and IDs you're looking for on a page. You can also get other information (eg CSS styles and JavaScript info) from DOM Inspector.

2 Error Console (Firefox Tools menu). This displays all script errors since you opened *Firefox* – hit Clear to get rid of them, then refresh the page with your non-functional script. If it's crashing, you'll get an error message. (Ignore the line number – it won't match your script because of the way in which user scripts are injected into the page – and just use the error message instead.)

3 Logging with the Greasemonkey GM_log function. These log messages will show up in the Error Console.