O'REILLY
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
http://www.onlamp.com/pub/a/onlamp/2005/04/28/kupu.html
See this if you're having trouble printing code examples

Rich Web Text Editing with Kupu

by Robert Jones
04/28/2005

Over the years that I've been writing web applications, I have always lacked a good way for users to enter and upload arbitrary blocks of text. A `TEXTAREA` tag in an HTML form handles basic text, but most of my applications have involved richer content. Having users write whatever they want in a word processor and then upload their documents to the server is a way to capture rich text, but it doesn't let you do much with the information once you have it. What I really needed was a WYSIWG editor that I could embed into an arbitrary web page. Now I've found one: Kupu.

Kupu is an open source application, written in JavaScript, that implements a flexible, full-featured HTML editor that runs in a web page without any special plugins. Its primary use is as an embedded editor in content management systems (CMS), like Zope or Plone, where it allows users to create their own web pages. Its design is flexible enough so that you can embed it into pretty much any web application without too much difficulty.

Without too much difficulty, that is, once you've figured out how it works. Like many other wonderful pieces of software, Kupu does itself a disservice with limited documentation and minimal or no comments in the source code. I understand why this happens. The developers want to use their limited time to write new code rather than create documentation. However, in doing so they severely limit the impact their work will have on the community. Open source software is not just about availability, it's also about accessibility.

In this article I want to make Kupu more accessible and show how easy it is to embed in your applications. It's a great application, and it deserves a wider audience.

JavaScript

Kupu was written by Paul Everitt, Guido Wesdorp, Philipp von Weitershausen, and colleagues, and it represents a remarkable feat of JavaScript programming. Speaking as one who struggles with that language for even the simplest of applications, I have the greatest admiration for the work that's gone into this project.

JavaScript seems to be experiencing a bit of a renaissance at the moment as people figure out the value of XMLHttpRequest. Kupu uses it, as does Google Maps, which has recently made its impressive debut, and Google Suggest. XMLHttpRequest is a way for the browser and server to communicate in the background without having to rebuild the entire page every time any data changes. Read more about that in Drew McLellan's article Very Dynamic Web Interfaces. Additionally, Joel Webber has dissected the technology used in Google Maps.

Unfortunately, using the full power of JavaScript, and XMLHttpRequest in particular, means that you run into compatibility issues with certain browsers. Right now, Kupu runs only on these browsers or higher versions thereof: Mozilla 1.3.1, Internet Explorer 5.5, and Netscape Navigator 7.1. It does not run on Konqueror, Opera, or Safari. Hopefully, upcoming releases will provide the missing features.

Installing Kupu

Basic installation is easy. Download the tar file from http://kupu.oscom.org/download/ (450K), and unpack it in a directory that is accessible from your web site. It creates and populates a directory called *kupu/*. You'll see a number of files and subdirectories. The beginner can happily ignore the vast majority of these. You don't need to run `make` or any of that

business. At this stage I suggest that you actively avoid the documentation in the *doc/* subdirectory.

To see what the editor looks like, start up a suitable browser and type in the path to this file on your web site `<your path>/kupu/common/kupu.html` (in other words, you supply everything before the *kupu/* directory name). You can look at an installation on my Kupu editor test page. You will see something like the page shown in Figure 1.

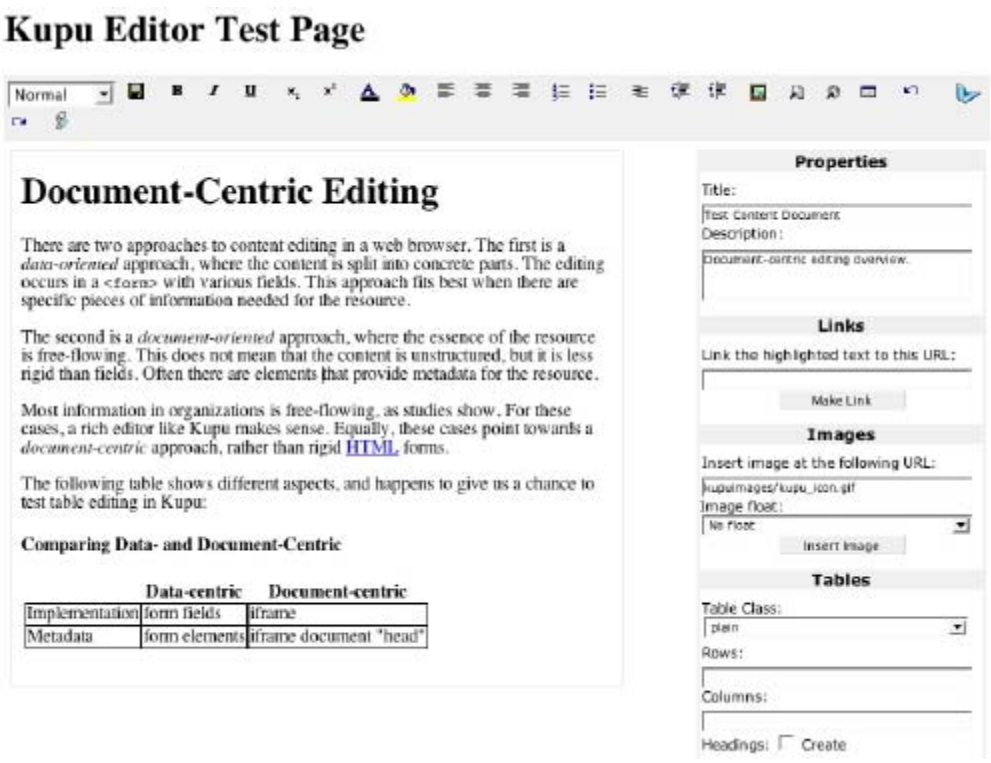


Figure 1. A Kupu example page

There are three parts to this page. The toolbar at the top contains icons for the standard functions people expect to find in an editor. The panel to the right contains blocks for certain functions that require additional information. The panel on the left is the actual document you are editing. In this case it preloads a default document that explains a little about the design philosophy of Kupu.

To edit the text, click somewhere in the text and start typing. If you are using Firefox and that doesn't do anything, press F7 on your keyboard. That enables something called *caret browsing*, which should fix the problem. Take it for a spin. The interface has a few quirks, but it's pretty intuitive. Select some text and change its color and format. Create some links. Insert an image from a URL and resize it. Best of all, insert a table and add and delete rows and columns. If you try to save your changes from this demo you will trigger an error, but I'll show how to fix that soon.

In its current release, Kupu may not do everything you want from a WYSIWYG editor. You can't change the fonts, for example, and you can only insert images that are already available on the Web. A couple of other features don't seem to work, at least in my hands. Its capabilities are more than enough for a lot of applications, though. Kupu is an amazing achievement when you consider that no plugin or Java applet is involved. This is just a web page with JavaScript.

Kupu's real value comes from its ability to be embedded in other applications. You can do this in one of two ways. The easiest is to have it upload its content to the server as a parameter to a CGI script, using the `POST` method. Less conventional, but the preference of the Kupu team, is to use the `PUT` method, along with a simple CGI script, to upload the content into a specific file on the web site.

Working with the Kupu Distribution

All of these features come at the price of a lot of JavaScript code in files that you need to include in pages that have Kupu

embedded in them. Even a basic page like *common/kupu.html* contains some pretty daunting code. What's more, all of those links to the included files are relative, which can cause all sorts of headaches if you want to mess around with the structure of the distribution directory. You really don't want to touch the distribution, but you need a way to link your custom content to it and still have everything work. Fortunately, there is a simple solution in the form of the `<base>` tag.

Place the distribution into a subdirectory under your web tree. Now create a directory elsewhere under the tree for your own pages. Copy over a template page (*kupu/common/kupu.html* or *kupu/common/kupuform.html*) and add a `<base>` tag in the `<head>` section of the web page that points to the *common* subdirectory in the distribution. For example:

```
<head>
<base href="http://www.craic.com/oreilly/kupu/kupu/common/">
</head>
```

This has the effect in your browser of prepending that URL to any relative links it encounters. Note that it has to be the complete URL path to the directory, including the host name, and that this will apply to all relative URLs in this page, not just the Kupu-related ones. Bear this in mind if you include any images, style sheets, or links to other pages on your site.

Using Kupu with `PUT`

The first approach for integrating Kupu is to use the `PUT` method to upload files to your web server. `PUT` is the neglected sibling of `GET` and `POST`, the most frequently used methods in the http protocol. It allows you to upload complete files onto a web server, placing them into a web tree with defined names. Of course you can do this using HTML forms and CGI scripts, but the intent of `PUT` is to simplify this process. Publishing Pages with PUT, from 1997, provides some background.

Perhaps the main reason that `PUT` has seen little use is its inherent insecurity. It has the potential to allow anyone to upload any kind of file to your site. You need to pay close attention to your configuration to close off any potential security hole. Making a simple error can create a major vulnerability. In contrast, you can think of using the `POST` method as starting out with a closed, secure system to which you explicitly allow access through your CGI scripts. It involves more programming effort, but the risk is easier to manage.

A `PUT` request sends two header lines ahead of the real content. `CONTENT_LENGTH` tells the server how much data to expect. `PATH_TRANSLATED` gives the path of the file into which to write that data. Think of this as the reverse of a browser fetching a page from the server. The default Apache httpd configuration accepts `PUT` requests, but in order for it to work you need to specify a CGI script that will handle the file. Do so with this directive, replacing the script name with your own:

```
Script PUT /cgi-bin/kupu/handle_put.cgi
```

Here is a simple CGI script that implements this. Note that this has no security checks built into it. Anyone could upload any file to a site that runs this script. Add your own restrictions before you install this on a public web site. You have been warned! (This example is *not* a live application on my site.)

```
#!/usr/bin/perl -w
# handle_put.cgi
# Basic PUT Handling routine with NO SECURITY!

if($ENV{'REQUEST_METHOD'} ne 'PUT') {
  errorMsg("Request method is not PUT");
}

my $filename = $ENV{'PATH_TRANSLATED'};

if(not $filename) {
  errorMsg("PATH_TRANSLATED was empty");
}

my $length = $ENV{'CONTENT_LENGTH'};
```

```
if(not $length) {
  errorMsg("CONTENT_LENGTH was empty");
}

# Add Security Checks Here! Limit access to certain
# directories and limit size and/or type of file.  E.g.

if($length > 100000) {
  errorMsg("CONTENT_LENGTH is too large");
}

# Read in the uploaded data
my $content = '';

my $nread = read(STDIN, $content, $length);
# Make the output more readable by adding newlines
$content =~ s/\>\</\>\n\</g;

# Write the file on the web server

open OUT, "> $filename" || errorMsg("Unable to open $filename");
print OUT $content;
close OUT;

# The 204 code signals the transfer was OK but does not
# update the current page - so you stay in the editor
print qq[Status: 204\n];
print qq[Content-type: text/html\n\n];
print $content;
exit;

sub errorMsg {
  my $msg = shift;
  print qq[Content-type: text/html\n\n];
  print qq[<html><head><title>Error</title></head>\n];
  print qq[<body>\nError: $msg<br>\n</body></html>\n];
  exit;
}
```

That is all the server needs. Now set up Kupu to use `PUT`. You don't need to specify the CGI script in the web page, as the Apache config file has preconfigured this. All you need to specify in the page is the filename on the server at which to write the content. Make a copy of *kupu/common/kupu.html* into your working directory and add a `<base>` tag, as described above, so that it can load in the JavaScript libraries from the distribution directory.

Take a look at the source of this page. With more than 300 lines, it's not for the faint of heart. What's more, a lot of it does not look like regular HTML. Fortunately, only two of these lines matter now, one near the top of the page and one near the bottom. As for the rest, leave it untouched for now. Specify the page to load into the editor on startup in the `<iframe>` at the bottom of *kupu.html*. Here I have it set to a blank page (*kupublank.html*):

```
<iframe id="kupu-editor" frameborder="0" src="kupublank.html" scrolling="auto">
```

Next, specify where to write the completed page on the server. Look for the `<kupuconfig>` block, about 30 lines down from the top of the page:

```
<kupuconfig>
<dst>http://www.craic.com/oreilly/kupu/no_such_file.html</dst>
```

Put the complete URL of the target file between the `<dst>` tag pairs. This directory needs to be writable by the web server, typically user `apache`. Note that you are specifying a single file. In these examples, you don't have the option of defining the filename at runtime.

Load the editor that you just configured into your browser, enter some text, and play around with the formatting. To save the file, click on the disk icon in the Kupu toolbar. It will appear that nothing has happened, as Figure 2 shows.

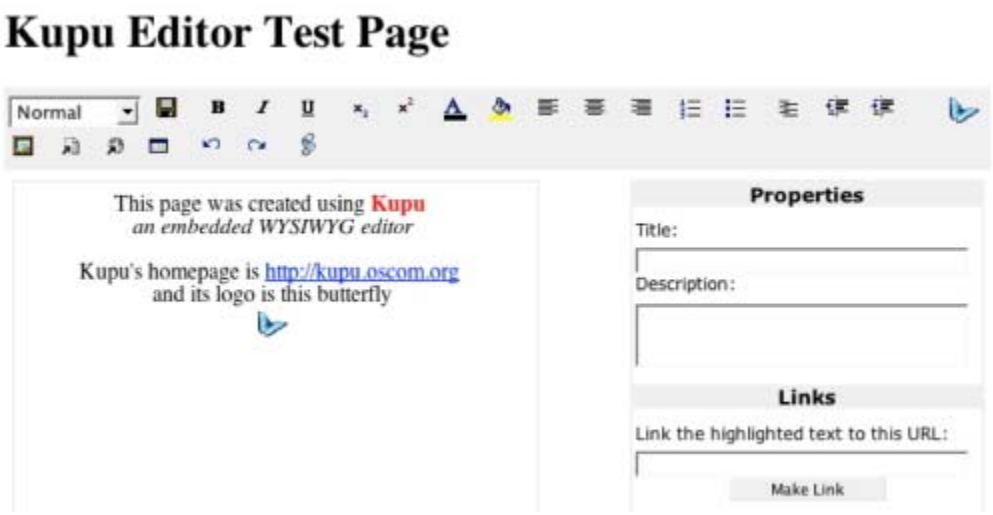


Figure 2. Freshly edited text

Now go to the URL that you defined as the target. You should see your newly defined page exactly as you defined it, but without all the toolbars. Figure 3 shows an example.

Figure 3. The newly created page

This is a simple example, and you will quickly see several shortcomings. You overwrite the target file every time you save, for example. Although the Kupu developers favor the `PUT` method, if you embed the editor into your own applications, especially a CMS, you will probably find that using it within a form will offer more control.

Using Kupu Within an HTML Form

The second approach is to wrap a form around the Kupu code. This requires more effort on the back end, as it needs a CGI script to handle the data. However, it gives you more control over what you do with the data than did the `PUT` technique, which simply mirrored what was in the editor.

Start with an example HTML form called *kupuform.html*, which you can find in the *kupu/common/* directory. In a browser, the page looks the same as the *kupu.html* from the previous example. The source for the page is very similar but with some critical differences. About 20 lines down from the top, you should find a `<form>` tag:

```
<form action="http://debris.demon.nl/printpost" method="POST">
```

The URL defines the script that will process the uploaded content. Replace it with your own script. Now go to the bottom of the file and look back eight or so lines for an `<iframe>` tag:

```
<iframe id="kupu-editor" frameborder="0"
       src="fulldoc.html" scrolling="auto">
```

The `src` attribute specifies the document to load into the editor when it starts. In this case, it's the one shown in the screen dump, called *fulldoc.html*.

To make this form your own, change these lines. First of all, make a copy of *kupuform.html* in your working directory, and add a `<base>` tag so you can access the JavaScript libraries as before. Start with a blank page in the editor by specifying an empty file in the `<iframe>` tag, using the file *kupublank.html* in the distribution directory.

You need to have a CGI script on the server that can do something with the uploaded data. Here is a simple script (*kupu_echo.cgi*) that reads the content posted by the form, in the parameter `kupu`, and gives it straight back to the user. Kupu generates HTML with no newlines, which makes viewing the source difficult, so the script inserts a newline character after each tag.

```
#!/usr/bin/perl -w
# kupu_echo.cgi

use CGI;
my $cgi = CGI->new();

my $text = $cgi->param('kupu');
$text =~ s/\>/\>\n/g;

print $cgi->header();
print $text;
```

Put this into the *cgi-bin/* directory on your web site, make it executable with `chmod a+x kupu_echo.cgi`, and put the URL for it into the `<form>` tag. The two tags to change should now look something like this:

```
<form action="http://www.craic.com/cgi-bin/kupu/kupu_echo.cgi" method="POST">

...

<iframe id="kupu-editor" frameborder="0"
       src="kupublank.html" scrolling="auto">
```

Give it a try. Go to *kupuform.html* and you should see a blank panel in the editor. Add some text and format it to your tastes. Click on the disk icon in the toolbar to upload it to the CGI script. If you need instant gratification, see the blank Kupu editing demo on my site.

That wasn't too painful. You had to change two lines in the template web page and write a small CGI script. Most importantly, you didn't have to touch a single line of JavaScript! This is a simple example, of course, but it gets you through the first phase in the learning curve.

A Larger Example

For a slightly more involved example, I've written a very basic blogging application using Kupu within a form as the way to create new records.

It consists of a template HTML file, which contains all the Kupu code, and a CGI script that modifies this on the fly, handles new entries submitted to the blog, and displays the titles of existing entries and the links to them. It's pretty basic, but it shows one way to integrate the editor into a form with other data-entry widgets. It also serves as a gentle introduction to modifying the Kupu template page.

Figure 4 is a screenshot of the application with a few entries already added to the blog. To make it realistic, I've tried to make the entries match the exciting content of those in most of the blogs out there today.

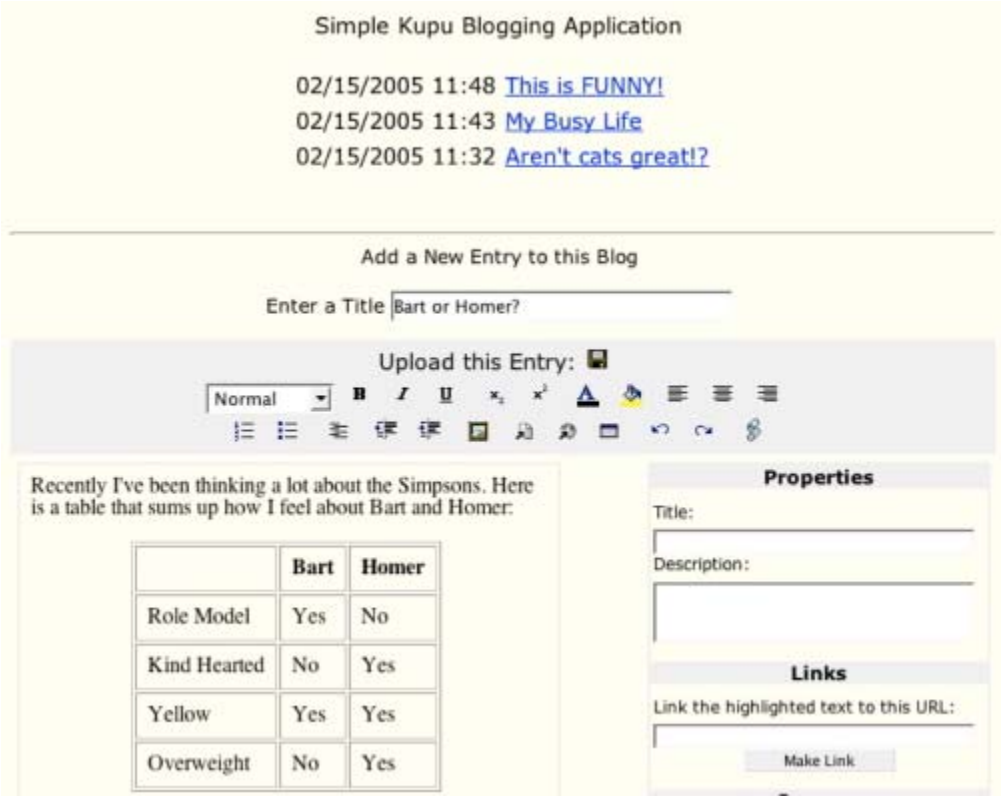


Figure 4. Kupu as a weblog editor

In addition to the editor, there is a regular text `<input>` widget used to enter a title for the entry. Using a form lets you combine your own data-entry widgets with the full power of the editor. I've made some changes to the template page so as to rearrange the Kupu toolbar a little. Look at some of the screenshots on the Kupu homepage for examples of how different CMSes have customized. To really get into that, you need to understand the JavaScript that makes up the application and come to grips with the XML it uses to configure itself. That is far beyond the scope of this article, however.

The complete source code of this application is too long to include here, but you can retrieve it as well as the other code examples from my Kupu site.

View the source of my template Kupu page and you will see comment lines of the form `<!-- ### -->` that indicate where I have modified the original *kupuform.html* template. The CGI script replaces any Perl-like variables in the template, like `$srcFile`, on the fly. (Yes, I know this would be easier in PHP!) I've also included the source for the Perl CGI script.

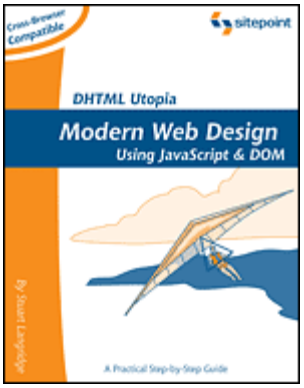
You can also try the simple Kupu-based blogging application on my site. (I delete all entries on a regular basis.)

Final Thoughts

The examples shown here merely scratch the surface of the Kupu software, but they lay the foundation for your own experiments and, for the brave among you, for understanding the XML and JavaScript magic that makes Kupu work. The technology used here and in other applications that use XMLHttpRequest is an important emerging force in web development. It's well worth your attention.

Robert Jones runs Craic Computing, a small bioinformatics company in Seattle that provides advanced software and data analysis services to the biotechnology industry. He was a bench molecular biologist for many years before programming got the better of him.

Related Reading



DHTML Utopia: Modern Web Design Using JavaScript & DOM
By Stuart Langridge

Return to ONLamp.com.